AUTOMATED MIXED-SIGNAL VERIFICATION USING MONITORS AND SIMULATION RELATIONS

Debjit Pal

Automated Mixed-Signal Verification using Monitors and Simulation Relations

Thesis submitted to

Indian Institute of Technology, Kharagpur

For the award of degree

of

Master of Science

by

Debjit Pal

Under the guidance of

Prof. Pallab Dasgupta
Dept. of Computer Science and Engineering
and
Prof. Siddhartha Mukhopadhyay
Dept. of Electrical Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
January 2013

©2013 Debjit Pal. All rights reserved.

To my Dear Parents and Teachers....

APPROVAL OF THE VIVA-VOCE BOARD

/ /20

Certified that the thesis entitled Automated Mixed-Signal Verification using Monitors and Simulation Relations submitted by DEBJIT PAL to Indian Institute of Technology Kharagpur, for the award of the degree of Master of Science has been accepted by the external examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

Signature: Name: (Member of DSC)	Signature: Name: (Member of DSC)	Signature: Name: (Member of DSC)

Signature: Name: (Supervisor)	Signature: Name: (Supervisor)
Signature: Name: (External Examiner)	Signature: Name: (Chairman)

CERTIFICATE

This is to certify that the thesis entitled **Automated Mixed-Signal Verification using Monitors and Simulation Relations**, submitted by **Debjit Pal** to Indian Institute of Technology Kharagpur, is a record of bona fide research work under our supervision and is worthy of consideration for the award of the degree of Master of Science of the Institute.

Prof. Pallab Dasgupta
Supervisor

Prof. Siddhartha Mukhopadhyay
Supervisor

DECLARATION

I certify that

a. the work contained in this thesis is original and has been done by me under the guidance of my supervisors.

b. the work has not been submitted to any other Institute for any degree or diploma.

c. I have followed the guidelines provided by the Institute in preparing the thesis.

d. I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

e. whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Debjit Pal

Acknowledgement

At the end of my pursuit towards obtaining Master of Science from IIT Kharagpur, when I look back I am simply overwhelmed by the amount of support, inspiration and confidence I gathered from a lot of people around me. I am just worried by the fact that in my effort to acknowledge my indebtedness towards them I might forget to mention a a few. However, I shall not shy away from the task and do my best. Firstly, I must thanks my family, especially my ever caring parents, who throughout my life have lent their unconditional material and emotional support to me. Whenever I lost my way they were there to help me out. I humbly dedicate this thesis to them. Being the younger child of the family, I was pampered a lot by my sister and later by my brother-in-law. They have given me everything without ever asking for it.

I would like to express my gratitude towards my supervisors Prof. Pallab Dasgupta and Prof. Siddhartha Mukhopadhyay for helping me in building this thesis. They give me suitable research problems and then guided me in solving those. The invaluable technical suggestions given by them certainly helped to complete this thesis in more technically correct form. It is due to Prof. Dasgupta's encouragement and support which helped me to learn numerous industry standard CAD tools with hands-on-experience. Also, I would like to express my gratitude towards Prof. Dipankar Sarkar for innumerable number of technical and non-technical discussion sessions I had with him. Their guidance, support, inspiration and constant encouragement gave me the will to overcome the failures and proceed towards my goal.

I would like to thank all of my friends at IIT Kharagpur for their co-operation which made my stay here enjoyable and memorable. I thank my co-workers Antara Di, Dr. Scott Little (formerly in Freescale Inc. and now in Intel, Portland) and Santhosh for helping me in different project works. I also take the pleasure of acknowledging Soumyadip (Pagol), Chandan Da, Aparna Baudi, Soumyojit Da (Robo Da), Kunal, Partha, Gargi, Devleena, Debashis Da (DMon Da), Pradipta Da (PD), Jyotirmoy Da (Moy), Sourish Da, Anshuman Da, Rajdeep Da, Satya Gautam, Kamalesh Da, Prasenjit Da, Aritra Da, Srobona Di, Pradipta, Jayeeta and my other numerous friends in VSRC for making IIT KGP a wonderful place to work at. I specially thank Arijit Da for giving me my initial lessons on Linux which helped me a lot later. Also, I have learned a whole gamut of things from Subrat Da during my initial phase of MS work. I would like to thank Anirban Da (VLSI Lab), Shibu Da (CSE Software Lab) for their technical support. I would like to thank "Cadence India Support" for helping me a lot while I faced problems in CAD tools. I would also like to thank Mr. John Gough (Design Manager, NSC UK pvt. Ltd.) for allowing me to do an internship in the Greenock Design center of NSC in 2009. I would also thank Semiconductor Research Corporation to partially support my MS work.

Finally, I want to thank almighty for his blessings without which the journey of my MS course would not have been so smooth.

Debjit Pal

List of Abbreviations

OVL	–	Open Verification Library
PORV	–	Predicate Over Real Variables
MTL	–	Metric Temporal Logic
OVA	–	Open Vera Assertions
MITL	–	Metric Interval Temporal Logic
STL	–	Signal Temporal Logic
LTL	–	Linear Temporal Logic
PSL	–	Property Specification Language
SVA	–	System Verilog Assertion
VLSI	–	Very Large Scale Integration
HDL	–	Hardware Description Language
VHDL	–	VLSI Hardware Description Language
LDO	–	Low Dropout
AMS	–	Analog and Mixed Signal
AMS-VL	–	Analog-Mixed Signal Verification Library
PLL	–	Phase locked Loop
PWM	–	Pulse Width Modulation
PFM	–	Pulse Frequency Modulation
BDD	–	Binary Decision Diagram
FSM	–	Finite State Machine
KL	–	Kanellakis-Smolka
PT	–	Paige-Tarjan
LTS	–	Labeled Transition System

List of Symbols

$\wedge$	–	Logical AND
$\vee$	–	Logical OR
$\neg$	–	Logical NOT (Negation)
$\times$	–	Cartesian Product
$\in$	–	Belongs to
$\preceq$	–	simulates
$\models$	–	models
$\Rightarrow$	–	implies
$\subseteq$	–	Subset of
$\rightarrow$	–	maps to
$\forall$	–	for all
$\exists$	–	there exists
$=$	–	is equal to
$\neq$	–	does not equal
$\Rightarrow$	–	implies
$\mathbb{R}$	–	real set

Abstract

Modern Analog and Mixed Signal (AMS) circuits consist of both digital and analog components. The verification environment for such circuits is mixed-mode and verification methods that work seamlessly in mixed-mode environment are needed. Although industry standard optimized mixed-mode simulators that can simulate both analog and digital components are now available but advanced debugging paradigm employs formal properties and verification libraries which are still not ubiquitous in mixed signal designs although, their use is now standard for digital circuits. In any circuit design, an implementation is typically a refinement of the specifications. Hence, it is essential to check conformance between implementation and specification. In the AMS domain conformance can be defined in terms of specific properties involving a set of signals over time. This thesis presents a methodology by which a verification engineer can build a property verification network graphically from a set of basic property blocks. It also presents a formal method for conformance checking between digital controllers of AMS circuits and further extends to a feature based simulation trace based conformance checking methodology for AMS Circuits.

In this thesis we present the syntax and semantics for a library of parameterized modules such that these modules can be composed graphically to create a verification network repository for different complex AMS properties. Thus, we have brought together the advantage of graphical composition from Open Verification Library (OVL) available for digital circuits and dense real time signal handling capacity of Signal Temporal Logic (STL) essential for continuous signals. We also show that these library modules can be interfaced with suitable auxiliary functions and can be used to verify properties that cannot be captured ordinarily by assertion languages. We show that the library modules are capable of synchronizing with AMS simulators and can generate multiple threads of property verification to ensure that no potential match / fail is missed.

Equivalence checking and simulation relation finding is a very standard and well studied problem in the digital domain, where equivalence is defined in terms of logic. In the AMS domain, equivalence is relevant only w.r.t certain features. One of the main part of an AMS circuit is the digital controller. These digital controllers have analog interface. For these controllers the inputs are not only propositions but also Predicates over Real Variables (PORVs). In this work, we present a symbolic method to find simulation relation between such controllers.

Finally, in this thesis we use the concept of feature based equivalence to find conformance of two AMS circuits over their simulation trace. We use a set of auxiliary functions to compute special features (which may or may not be simple functions of time and are not available directly from temporal trace) from the simulation trace and monitor other properties with the help of modules from the verification library. The proposed approach has been demonstrated using a family of LDOs and BUCK Regulators as reference.

Keywords: *Assertion, Passive-Online Verification Methodology, Labeled Transition Systems, Simulation Relation, Feature Based Equivalence, Co-Simulation.*

Contents

Certificate of Approval vii

Certificate ix

Declaration xi

Acknowledgement xiii

List of Abbreviations xv

List of Symbols xvii

Abstract xix

1 Introduction 1
- 1.1 Motivation and Objectives . . . 2
- 1.2 Summary of Contributions . . . 3
  - 1.2.1 AMS Verification Library . . . 3
  - 1.2.2 Verification of Simulation Relations . . . 4
  - 1.2.3 Feature based Online Conformance Checking . . . 6
- 1.3 Organization of the Thesis . . . 7

2 Background and Literature Review 9
- 2.1 Assertions and Open Verification Library . . . 9
- 2.2 Logic Languages STL and AMS-LTL . . . 14
- 2.3 Equivalence and Simulation Relations . . . 16
- 2.4 Algorithm for Equivalence Checking . . . 21
  - 2.4.1 Kanellakis-Smolka Algorithm . . . 24
- 2.5 Concluding Remarks . . . 25

3 AMS Verification Library 27
- 3.1 Definitions and Preliminaries . . . 29
- 3.2 The Structure of AMS Verification Library . . . 30
  - 3.2.1 CaptureAndHold . . . 31
  - 3.2.2 GenerateDelay . . . 32
  - 3.2.3 ArithmeticOperator . . . 32
  - 3.2.4 EventDetector . . . 33
  - 3.2.5 EventDetector_Extended . . . 35
  - 3.2.6 PredicateEvaluator . . . 36
  - 3.2.7 PredicateEvaluator_Extended . . . 38

3.2.8 BoolOperator 39
3.2.9 GlobalOperator 40
3.2.10 EventuallyOperator 43
3.2.11 UntilOperator 45
3.2.12 PredicateAssert 48
3.3 Representative Verification Networks with AMS-VL Components 49
3.4 Tool Flow and Implementation Issues 55
3.4.1 Synchronization with the AMS Simulator 56
3.4.2 Spawning Threads for Overlapping Matches 59
3.5 Simulation Results 62
3.6 Concluding Remarks 63

4 Verification of Simulation Relations 65
4.1 Simulation Relation Finding Methodology 69
4.1.1 Formal Model of Computation 69
4.2 Methodology and Tool Flow to find Simulation Relation 76
4.2.1 Pre-Process Steps 76
4.3 Concluding Remarks 82

5 Feature based Equivalence Checking with AMS-VL 83
5.1 Different topologies for Online Conformance 84
5.2 Examples of Conformance Checking Networks 86
5.2.1 Example of Topology-I 86
5.2.2 Example of Topology-II 87
5.2.3 Example of Topology-III 87
5.2.4 Example of Topology-IV 89
5.3 Concluding Remark 91

6 Conclusion 93
6.1 Summary of Achievements 93
6.2 Future Work 94

Appendices 95

A Sample Auxiliary Function Modules 97
A.1 Auxiliary Module to model Start-Up of BUCK and LDOs 97
A.2 Auxiliary Module to measure Frquency of PLL 100

B Testcases and Sample Properties of AMS-VL 103
B.1 Low Dropout Regulator (LDO) 103
B.2 Voltage Mode Controlled BUCK Regulator 104
B.3 Sample Properties for Low Dropout Regulators 106
B.4 Sample Properties for BUCK Regulators 108
B.5 Sample Properties for Integrated Power Management Unit 109

Bibliography 113

List of Figures

1.1 Tool Flow of AMS-VL 5
1.2 A Specification and Two Implementations 5
1.3 Tool Flow for Feature based Conformance Checking 7

2.1 TS_2 simulates TS_1 23
2.2 Before applying Kanellakis-Smolka's Algorithm 24
2.3 After applying Kanellakis-Smolka's Algorithm once 25

3.1 Temporal Trace of EventDetector 35
3.2 Temporal Trace of EventDetector_Extended 37
3.3 Temporal Trace of PredicateEvaluator 38
3.4 Temporal Trace of PredicateEvaluator_Extended 40
3.5 Temporal Trace of GlobalOperator 42
3.6 Temporal Trace of EventuallyOperator 44
3.7 Temporal Trace of UntilOperator 47
3.8 Different Cases of PredicateAssert Module 50
3.9 EventDetector Deglitched Module 52
3.10 Multiple Event Detection 52
3.11 AMS-VL Realization of Example 3.3 53
3.12 AMS-VL Realization of Example 3.4 54
3.13 AMS-VL Realization of Example 3.5 55
3.14 Tool Flow of AMS Verification Library 55
3.15 Schematic of Example 3.3 56
3.16 LDO Test Case and associated Verification Networks 57
3.17 Timing Diagram of Example 3.6 58
3.18 Scenario for Property Checking in Parallel Threads 59

4.1 A Specification and Two Implementations 68
4.2 Specification LTS Annotated with Refinement Directives γ 70
4.3 Zone of Interest of Implementation PORVs 71
4.4 Paths in Specification and Implementation LTS 72
4.5 Pre-Process Step 1 77
4.6 Pre-Process Step 2 78
4.7 Simulation Relation Finding Tool Flow 81

5.1 Conformance of Two AMS Models / Circuits 83
5.2 Block Diagram of Topology-I 84
5.3 Block Diagram of Topology-II 85
5.4 Block Diagram of Topology-III 85
5.5 Block Diagram of Topology-IV 86

5.6 Monitoring Network for Example 5.1 87
5.7 Monitoring Network for Example 5.2 88
5.8 Monitoring Network for Example 5.3 88
5.9 Monitoring Network for Example 5.4 89
5.10 Schematic of Example 5.4 90

B.1 Block Diagram of an LDO Regulator Circuit [59]. 104
B.2 Output Voltage of LDO Regulator in Different Modes of Operation. . 105
B.3 Block Diagram of a Buck Regulator Circuit [47]. 105
B.4 Output Voltage of Buck Regulator Circuit [47]. 106

List of Tables

3.1 Different Parameters and their Context of Utilization 30
3.2 Broad Classification of AMS-VL Modules 31
3.3 Ports of CaptureAndHold Module 32
3.4 Parameters of CaptureAndHold Module 32
3.5 Ports of GenerateDelay Module 32
3.6 Parameters of GenerateDelay Module 32
3.7 Ports of ArithmeticOperator Module 33
3.8 Parameters of ArithmeticOperator Module 33
3.9 Ports of EventDetector Module 33
3.10 Parameters of EventDetector Module 34
3.11 Ports of EventDetector_Extended Module 35
3.12 Parameters of EventDetector_Extended Module 36
3.13 Ports of PredicateEvaluator Module 37
3.14 Parameters of PredicateEvaluator Module 37
3.15 Ports of PredicateEvaluator_Extended Module 38
3.16 Parameters of PredicateEvaluator_Extended Module 39
3.17 Ports of BoolOperator Module 40
3.18 Parameters of BoolOperator Module 40
3.19 Ports of GlobalOperator Module 41
3.20 Parameters of GlobalOperator Module 41
3.21 Ports of EventuallyOperator Module 43
3.22 Parameters of EventuallyOperator Module 43
3.23 Ports of UntilOperator Module 46
3.24 Parameters of UntilOperator Module 46
3.25 Ports of PredicateAssert Module 48
3.26 Parameters of PredicateAssert Module 48
3.27 CPU Time for Simulations of Circuits 62
3.28 Description of the Testcases 63

Chapter 1
Introduction

Modern Analog and Mixed Signal (AMS) circuits consist of both digital and analog components. The verification environment for such circuits is mixed-mode and verification methods that work efficiently in mixed-mode environment are needed. Although, industry standard optimized mixed-mode simulators [5] that can simulate both analog and digital components have become available, but advanced debugging constructs such as formal properties [48, 49, 50, 51, 52] and verification libraries are still not ubiquitous in mixed signal designs.

In the last decade, there have been two significant directions towards improving the state-of-art for mixed-signal verification. These are, (a) the use of behavioral modeling and (b) the use of assertion and verification libraries. Behavioral models have been found to be useful at various stages of AMS design flow. For example, application engineers use behavioral models for demonstration purposes, designers use them for design space exploration, verification engineers use behavioral models for developing the environment for designs under test (DUTs), system engineers use behavioral models for system level debugging. The need to maintain a certain level of conformance / equivalence between these models is motivation for studying equivalence checking methods in this domain.

The need for assertions and verification libraries in mixed-signal verification and debugging has been acknowledged at various fora – in the industry as well as the academia. Accellera [1], the industry consortium responsible for developing previous assertion languages such as SystemVerilog Assertions (SVA) [6] and Property Specification Language (PSL) [7], have started an initiative towards developing the language standards for specifying mixed-signal assertions in a formal way.

There are several important differences between formal properties for the digital domain and those for AMS domains. In the digital domain, properties are expressed over Boolean signals and evaluated at well defined clock boundaries. On the other hand, in the AMS domain, the signals of interest (such as voltages and currents on various nets) are real valued and events can occur anywhere in

time (that is, time is dense). Therefore integration of AMS properties into the core fabric of AMS design verification has serious synchronization challenges between the simulator and the property monitors.

Recently, a significant volume of research on assertion verification for mixed-signal designs has been reported in [50, 51, 52]. These also led to the development of prototype tool kits that integrate with standard AMS simulation platforms, thereby augmenting them with assertion monitoring capabilities similar to the digital EDA platform.

The focus of this thesis is on developing methods to meet some of the verification requirements stated above. The thesis presents new abstractions and algorithms for checking conformance between the specification and an implementation of controllers for hybrid systems. The thesis also present a verification library that can be used to monitor complex AMS behaviors and can be used also to monitor feature based conformance between AMS models over simulation.

Section 1.1 presents the motivation of this work. Section 1.2 presents a summary of the contributions, followed by technical outlines of these contributions. Section 1.3 presents the organization of the thesis.

1.1. Motivation and Objectives

It has been observed that assertions alone are not expressive enough to capture complex AMS behaviors. Often it is easier to transform the signals (through auxiliary functions) to a different domain and express the desired property succinctly over the transformed signals [51]. This approach requires a library of auxiliary functions. Most verification engineers, without extensive training and experience, find it difficult to use the appropriate combination of auxiliary functions and assertions to develop the specifications for complex AMS properties. Further, with little or no experience in linear temporal logic (LTL) [57] and signal temporal logic (STL) [48], which forms the core fabric of AMS property languages like AMS-LTL [52], most verification engineers find it difficult to write complex temporal properties in above mentioned assertion languages. Hence, there is a need to construct a repository of modules performing basic LTL operations and modules for auxiliary functions which can be interconnected graphically to express complex behavioral properties of AMS circuits. One of the objectives of this thesis is to address this requirement.

In any design flow, the implementation is a refinement of the specification, that is, the set of behaviors admitted by the implementation is a subset of the behaviors admitted by the specification. Equivalence checking and simulation relation finding is a very standard and well studied problem in the digital domain where equivalence is defined in terms of logic and and there exists a rich body of

literature on computing sequential equivalence [23, 24, 26, 44, 45] and simulation relations [40, 58] between finite state machines. In the AMS domain, conformance is relevant with respect to specific *features*, ranging from simple region containment predicates to very complex behaviors (not necessarily in the time domain). A very important component of modern hybrid systems (like AMS circuits) is the digital controller which interacts with analog components and actuates their behaviors. Such controllers form an important class of structures that can have predicates over real variables (PORVs) as inputs in addition to propositions. It is important to formally verify whether the implementation of a controller is a refinement of the model of its specification. Another objective of this thesis is to study formal simulation / refinement relation for such controllers.

The objective of this thesis can therefore be articulated in terms of the following problems, namely (a) to design a library of modules which can be used to compose graphically a wide range of property monitors of varying complexity with ease and can be interfaced with auxiliary functions seamlessly, (b) to propose a method to formally compute simulation relation between abstractions of controllers for hybrid systems and (c) to leverage the verification monitors to check feature based conformance between two AMS circuits or models during simulation.

1.2. Summary of Contributions

The thesis presents the following three contributions :

- A verification library consisting of basic parametrized modules which can be connected graphically to compose property monitors for verification of AMS circuits over simulation.
- A formal symbolic method to find simulation relation between predicate labeled abstraction of digital controllers for hybrid systems.
- A method for feature based conformance checking between two AMS circuits / models over simulation online, leveraging the verification library and auxiliary modules.

The following three subsections outline the major aspects of these contributions.

1.2.1. AMS Verification Library

This work presents the genesis of a library of checker modules that can be connected graphically to build verification networks for complex properties of AMS circuits. In this approach we have brought together the advantage of graphical composition of verification networks from Open Verification Library (OVL) [2] of

the digital domain and the real time signal handling capacity of Signal Temporal Logic (STL) [48, 49] to create a toolbox for verification of AMS circuits over a simulation trace. We call it a *Passive Online* verification library - (a) *online* because the checkers check the simulation trace online (that is, as soon as it is generated by the simulator) and (b) *passive*, because the verification network built from the checkers do not modify the behavior of the circuit or the testbench. The contributions of this approach are as follows :

1. We have proposed a library of parameterized modules which can be interconnected graphically on a schematic to create verification networks for properties of AMS Circuits. A glimpse of the modules is given in the Table 3.2. We have shown that the library modules can be interfaced easily with suitable auxiliary functions for verifying AMS properties in the time domain.
2. The proposed library modules handshake with AMS simulators and can place additional simulation points near the events of interest. We have shown that the overhead incurred during simulation for adding these extra modules for verification is of the order of 7%-10% for large industrial test cases like Buck regulators and integrated circuit netlist. The large simulation time for these AMS circuits always supersedes the additional time incurred due to the inclusion of the AMS-VL modules.

Some features of the library are :

1. It has been shown that not only time domain properties but frequency domain properties can be verified with the library modules with the help of auxiliary functions. The modules can be treated as parameterized black boxes in the graphical verification framework by adjusting different parameters (as shown in Figure 3.13).
2. The library modules can be used to capture certain properties which are difficult to express in logic languages like LTL and STL as they can face explosion in the length of formula. Also, we have shown that the AMS-VL monitor network can contain cycles for expressing interesting properties.

Figure 1.1 shows the tool flow for AMS-VL.

1.2.2. Verification of Simulation Relations

In a multi-stage design cycle, it is the usual practice to check conformance between the *golden model* (hereby referred as *specification*) and the implementation. For robust satisfaction of *specification* by *implementation* it is always required that all behaviors of *implementation* are contained in the admissible *specification* behaviors. This work presents a symbolic simulation relation finding algorithm

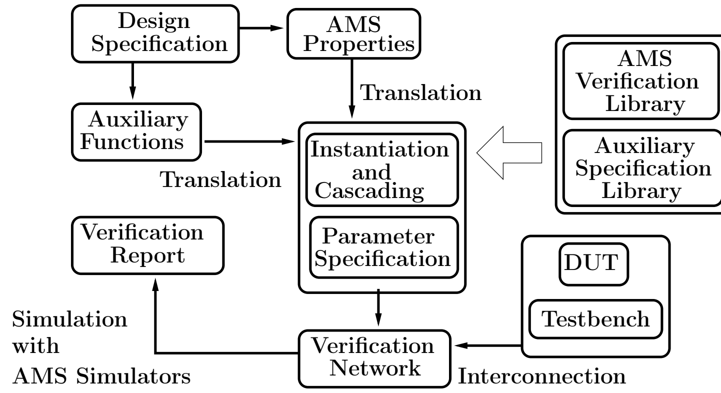
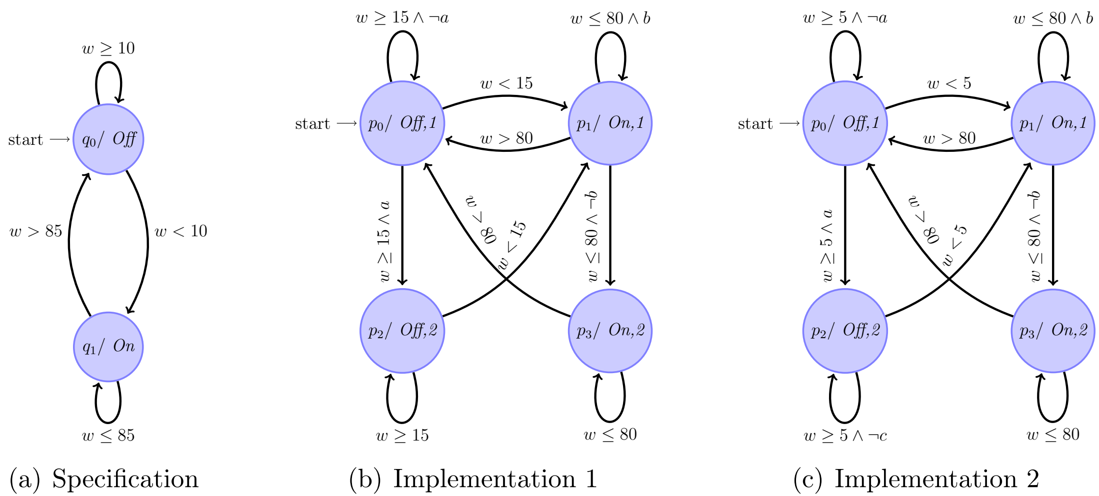


Figure 1.1: Tool Flow of AMS-VL



(a) Specification (b) Implementation 1 (c) Implementation 2

Figure 1.2: A Specification and Two Implementations

between *implementation* and *specification* controllers of hybrid systems. We have extended the classical algorithm of Kanellakis-Smolka (KS) [44, 45] for finding simulation relation to our problem domain. In Figure 1.2, we show one specification and two candidate implementations for a water level controller automaton. Implementation of Figure 1.2(b) satisfies the design intent of Figure 1.2(a) more robustly with some tolerance for aberrations in reading the water level but the implementation of Figure 1.2(c) is not acceptable. *Implementation*-1 never allows the water level to go beyond the safe limits of 10 units and 85 units as required in the design intent. But *Implementation*-2 allows the water level to go as low as 5 units while going from *Off* to *On* state. Hence, *Implementation*-1 is acceptable but *Implementation*-2 is not. Our objective is to formally define conformance such that the first controller implementation is decided to be correct and the second is not. Therefore we require techniques for finding *simulation relations* over predicate labeled transition systems. As demonstrated in the above water controller

example, the Predicates Over Real Variables (PORVs) used in the specification (like $w < 85$, $w > 10$) and in the implementation (like $w < 80$, $w > 15$) are not necessarily same, although they are defined over the same set of real variables.

An *implementation* is never a verbatim translation of *specification*. An *implementation* interacts with many non-ideal situations which a *specification* does not account for. Hence, to make sure that *implementation* never violates *specification* even under strongest non-ideal situations, for the *specification* controller we need a *refinement directives* that indicates the admissible directions in which input PORVs can be *strengthened* or *relaxed* in *implementation* to cope with non-ideal environments. For example, in Figure 1.2(a), the PORV labeling the transition from *Off* to *On* state i.e $w < 10$ can be *weakened* to $w < 15$ in an implementation (to allow aberration for reading real valued variable w) but the PORV $w \geq 10$ labeling the self loop at the *Off* state can be *strengthened* to $w \geq 15$ as shown in Figure 1.2(b). This does not follow automatically from the automaton of Figure 1.2(a) and needs to be explicitly specified by the user.

In Chapter 4, we have proposed such a symbolic *simulation relation* finding methodology. We have shown the necessary transformation steps required to reduce the problem in hand so that we can use the KS algorithm.

1.2.3. Feature based Online Conformance Checking between AMS Circuits

In this work we propose a method for conformance checking of two AMS circuits / models over simulation. Every AMS circuit shows both continuous and discrete dynamics. Hence, the behavior of the circuit can be divided into some modes of operation over its terminal voltages and currents. There exists certain properties / features which uniquely characterize a mode of operation of an AMS circuit. These features need not to be temporal always and may need to be derived with some additional calculations over simulation trace with the help of auxiliary modules. In a multi-cycle design procedure, it is required that the refined design conforms to its predecessor w.r.t certain features rather than everywhere of its operation. Further as we explained in introduction, a certain level of conformance is necessary between the behavioral models of AMS circuits used by different engineers for different purposes.

For example, we have a behavioral model of Buck regulator in Verilog-AMS [3] which acts as a specification and an implementation of the same Buck regulator as a circuit. One of the feature based on which we seek conformance between the model and the circuit is the *switching frequency of oscillation in the PWM mode of operation*. We use the AMS-VL modules along with necessary auxiliary modules to translate such definition of conformance between two AMS models / circuits to

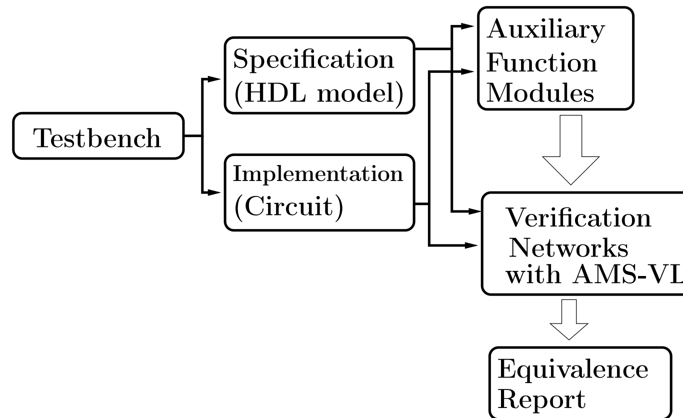


Figure 1.3: Tool Flow for Feature based Conformance Checking

a monitor network which can check the conformance online over simulation.

The proposed scheme has been shown in the Figure 1.3. The contributions are as follows :

1. We have defined several topologies to find feature based equivalence of two AMS circuits / models over simulation run. The reason, for which this kind of conformance is important in AMS domain, has been explained.
2. With the help of several examples we have explained the way AMS-VL modules and auxiliary modules can be used to translate the definition of conformance between two AMS models into a monitor network. The monitor network keeps checking the simulation trace continuously and generates the conformance report. Once constructed, such monitor networks can be stored in a repository for future re-use.

1.3. Organization of the Thesis

The rest of the thesis is organized in the following chapters. A summary of the contents of the remaining chapters are as follows :

Chapter 2 : This chapter contains a detailed study of the background required for the thesis and survey of relevant research.

Chapter 3 : This chapter presents a passive online verification methodology of AMS circuits with the help of proposed AMS-VL library and auxiliary functions. We explain the modules, their syntax and semantics, working of the modules with several examples, implementation issues and our approach to address those issues.

Chapter 4 : This Chapter presents a formal symbolic method for conformance checking of predicate labeled abstraction of digital controllers of

hybrid systems. We have explained the transformation steps required to map the proposed simulation relation finding problem to the simulation relation finding algorithms of digital domain.

Chapter 5 : This chapter presents a feature based online conformance checking methodology for two AMS circuits / models over simulation capitalizing the online monitoring capability of AMS-VL and auxiliary modules.

Chapter 6 : In this chapter, we summarize with the conclusions on the contributions of this thesis and list some possible future scopes of this work.

Chapter 2
Background and Literature Review

The primary aim of the chapter is to provide some background concepts that are necessary for the foundation of this thesis. This chapter gives a brief overview of the assertion languages and the verification library used in the digital domain verification. Also, it gives a glimpse of the assertion languages of AMS domain. It presents some relevant literature in the area of equivalence checking of AMS circuits and hybrid systems and describes a classical algorithm to find simulation relation in digital domain which will be the basis for our proposed algorithm to find simulation relation of the controllers of hybrid systems.

The chapter is organized as follows :

Digital domain assertion languages like LTL has been explained in the Section 2.1. The Open Verification Library has been explained in the same section. In the Section 2.2, AMS extensions of assertion languages like MITL, STL and AMS-LTL has been described with the help of suitable examples. Section 2.3 gives a brief description of the work reported in the domain of AMS circuit equivalence checking. Section 2.4 describes the classical simulation relation finding algorithm due to Kanellakis-Smolka in detail along with necessary terminologies.

2.1. Assertions and Open Verification Library

Assertions refer to the statements that have *Truth* associated with them. For example, the assertion *the output voltage of the circuit shall cross 3V* may have *true* or *false* value depending upon the prior knowledge one may posses about the circuit. One may associate the notion of time and can construct *temporal assertions*. For example, the above assertion can be modified in this way : *the output voltage of the circuit shall cross 3V within 10 μs after reset is disabled*. In this work we discuss assertions from the viewpoint of circuit designers - especially mixed-signal circuit designers.

Designs are made to meet certain specifications. Verification methods try to validate the designs against those desired specifications. But the inherent ambiguity of English language lead to mis-interpretation of the desired properties and

hence may lead to incorrect designs. One of the main aim of formal assertions is to avoid such ambiguities. This work focuses on methods that may be used to specify behaviors of systems involving both discrete and continuous variables / signals.

In digital domain, verification techniques are used for decades. There are two ways in which verification is carried out in digital domain :

1. Formal Verification

2. Dynamic Assertion based Verification

The former uses *formal verification* techniques in order to validate whether all possible execution of the system satisfies the specification whereas the later technique simulates the system with a particular testbench and validate certain *traces only*. Assertions are widely used in simulation based verification for monitoring complex temporal behaviors in digital integrated circuits. Assertions languages like *Property Specification Language* (PSL) [7] and *SystemVerilog Assertions* (SVA) [6] derive their syntactic fabric from temporal logics, like *Linear Temporal Logic* (LTL) [57]. Propositional temporal logic extend Boolean logic by allowing us to relate the truth of Boolean propositions in different time worlds. The syntax of LTL [57] is defined over a set of atomic propositions, $\mathcal{AP}$, as follows:

- Each $p \in \mathcal{AP} \cup \{\top\}$ is a LTL formula, where $\top$ denotes true.

- If f and g are LTL formulas, then so are $\neg f$, $f \wedge g$, $\mathcal{X} f$ and $f \mathcal{U} g$.

$\mathcal{X}$ represents the next-time operator and $\mathcal{U}$ represents the until operator. The formula $\mathcal{X} f$ is true in a time world iff f is true in the next time world. The formula $f \mathcal{U} g$ is true in a time world iff g is true in some future time world and f is true in all time worlds in between. LTL forms the backbone for assertion language standards like SVA and PSL adopted by the industry. The task of extending assertion languages towards capturing Analog and Mixed-Signal (AMS) behaviors is being seriously pursued by the research community [48, 49, 50, 51, 53] as well as industry consortia [1]. However, all known formal techniques goes into serious capacity limitations even for small AMS systems. In this chapter, we elaborate on different existing formal specification languages relevant for systems that involve both continuous and discrete signals.

The main task for property verification are as follows.

- To specify the design intent in terms of formal specification languages.

- To verify them on the implementation.

This chapter summarizes different existing formal specification languages that are relevant to specify systems involving continuous and discrete signals. We take the first example from the circuit domain.

Example 2.1 *: Consider the following specification of a circuit.*

1. *If the output of the circuit is greater than 3V and reset is asserted then eventually the output voltage will become less than 1V.*
2. *If input voltage is greater than 2V and enable is asserted, then eventually the output will become greater than 1.5V.*

We may consider a simple extension of LTL allowing relational Predicates Over Real Variables (PORV) [48, 53] as atomic propositions. This simple extension facilitate to express the requirements shown in Example 2.1 in the following way.

1. $(V_{out} > 3V) \wedge reset \Rightarrow \mathcal{F}(V_{out} < 1V)$
2. $(V_{in} > 2V) \wedge enable \Rightarrow \mathcal{F}(V_{out} > 1.5V)$

We can associate the notion of time with the temporal operators to come up with real time temporal specification.

Example 2.2 *Consider the modified properties from Example 2.1*

1. *If the output of the circuit is greater than 3V and reset is asserted for 10μs, then eventually the output voltage will become less than 1V.*
2. *If input voltage is greater than 2V and enable is asserted for 5μs, then eventually the output will become greater than 1.5V.*

Early attempts towards developing assertion languages for the AMS domain have largely been limited to the extensions which allow the use of PORVs [48, 53] and interpret the temporal operators according to the dense real time semantics as opposed to discrete real time semantics [17, 18, 19]. For example, a property which says that signal y is asserted between 3 to 5 time units of signal x being asserted may be written in Metric Temporal Logic (MTL) [18] as:

$$\mathcal{G}(x \Rightarrow \mathcal{F}_{[3,5]}\ y)$$

In MTL the signals x and y are Boolean and the future operator, $\mathcal{F}_{[3,5]}$ is interpreted over discrete time-steps, that is, it is expected that if x is high at time-step t, then y will be high in time-step $t+3$ or $t+4$ or $t+5$. Any transient

rise and fall of y between these time-steps will not be considered. On the other hand, consider an AMS property which says that if the voltage $V(in)$ of a battery charger is above 5V, then within 3 to 5 seconds the voltage $V(out)$ of the charger will be above 3V. This can be expressed by extending MTL with PORVs as follows:

$$\mathcal{G}((V(in) > 5V) \Rightarrow \mathcal{F}_{[3,5]} (V(out) > 3V))$$

In this property, $V(in) > 5.0$ and $V(out) > 3$ are PORVs which are natural extensions of atomic propositions like x and y in the previous property. Moreover, the future operator, $\mathcal{F}_{[3,5]}$ is interpreted over dense time which admits the fact that the truth of the PORV, $V(out) > 3$, may change transiently at any point of time between 3 and 5 seconds, and may not be visible at some clock boundary. In literature, several such dense time logics have been studied, which include Timed Propositional Temporal Logic (TPTL) [19], Metric Temporal Logic (MTL) [18], Metric Interval Temporal Logic (MITL) [17], to reason about Boolean signals over dense time. Analog Specification Language (ASL) was presented in [65] for verifying complex static and dynamic circuit properties like oscillation and start-up time. The authors of [30] proposed an extension of CTL to verify transient response of analog circuits offline.

The properties mentioned in Example 2.2 cannot be expressed succinctly with the help of the logic languages discussed so far. In Section 2.2, we show that with the help of Signal Temporal Logic (STL), an extension of MITL, the above properties can be encoded.

The Open Verification Library (OVL) [2] provides designers, integrators and verification engineers with a single, vendor-independent interface for design validation using simulation, hardware acceleration, formal verification and semi-/hybrid-/ dynamic-formal verification tools. It is interesting to note that the OVL was a reasonable popular alternative to the use of assertions in the digital domain. OVL was accepted into industrial practice primarily because mixed-mode simulation was in its infancy and therefore AMS simulators did not support assertion languages like Open Vera Assertions (OVA) [9] and SystemVerilog Assertions (SVA) [6]. Another reason for the acceptance of OVL was that verification engineers found it more convenient to graphically compose OVL modules to develop complex monitors as compared to developing assertions which capture the same behavior. The OVL is composed of a set of assertion checkers that verify specific properties of a design. These assertion checkers are instantiated in the design establishing a single interface for design validation. OVL assertion checkers are partitioned into the following classes:

- Combinational assertions - behavior checked with combinational logic.

- 1-cycle assertions - behavior checked in the current cycle.
- 2-cycle assertions - behavior checked for transitions from the current cycle to the next cycle.
- n-cycle assertions - behavior checked for transitions over a fixed number of cycles.
- Event-bound assertions - behavior is checked between two events.

Each OVL assertion checker has its own set of parameters. A few parameters common to all checkers are as follows: *severity_level*, *property_type*, *coverage_level*, *gating_type* etc.

An example of a 1-cycle assertion checker is *ovl_always*. The *ovl_always* checker checks its input - a single-bit expression *test_expr* at each active edge of the clock. If *test_expr* is not TRUE, a violation occurs and prints an error message.

An example of a 2-cycle assertion checker is *ovl_quiescent_state*. It has three inputs *sample_event*, *state_expr*, *check_value* other than *clock*, *reset* and *enable*. At every active edge of *clock* it checks whether the single bit expression *sample_event* has transitioned to TRUE i.e. FALSE on the previous clock edge and TRUE on the current clock edge. If so, it checks whether the current value of *state_expr*, equals that of *check_value*. If these values are not equal the assertion fails.

An example of an n-cycle assertion is *ovl_frame*. It ensures that when a specified start event is TRUE, then an *expression* must not evaluate TRUE before a *minimum* number of clock cycles and must transition to TRUE no later than a *maximum* number of clock cycles. A similar property checker is *ovl_width* which ensures that when value of an *expression* is TRUE, it remains TRUE for a minimum number of clock cycles and transitions from TRUE no later than a *maximum* number of cycles.

There are *event bounded* assertion checkers like *ovl_win_change* which ensures that the value of an expression changes in a specified window between a *start* event and an *end* event. The n-cycle and event bounded assertions are the ones which are very relevant in the Analog Mixed Signal domain.

OVL checkers are instantiated as modules in the digital designs. OVL has different variants for integration with Verilog, VHDL and SVA. Today, analog behaviors are not supported in OVL. We propose such a library of checkers following the line of OVL which can check properties over the analog behaviors. We describe such an approach in the Chapter 3.

2.2. Logic Languages STL and AMS-LTL

In [48] an extension of MITL [17], called Signal Temporal Logic (STL), was proposed which allows PORVs along with the dense timed temporal operators for specifying desired properties of continuous signals. The logic is based on a bounded subset of the real-time logic MITL, augmented with a static mapping from continuous domains into propositions. From formula in this logic the authors created automatically property monitors that can check whether a given signal of bounded length and finite variability satisfies the property. A prototype implementation of this procedure was used to check properties of simulation traces generated by Matlab/Simulink.

Definition 2.2.1 [Syntax of STL] *: An STL formula φ is defined inductively by the following grammar :*

$$\varphi ::= \top \mid p \mid \neg\, \varphi \mid \varphi \;\wedge\; \varphi \mid \varphi\, \mathcal{U}_I\, \varphi$$

We have $\mathcal{AP} \cup \mathcal{AP}_A$ as the finite set of the atomic propositions, where $\mathcal{AP}$ is the set of the Boolean propositions and $\mathcal{AP}_A$ is the set of PORVs. In the above grammar, $p \in \mathcal{AP} \cup \mathcal{AP}_A$ and I is an interval. If $X = \{x_1, x_2, \ldots, x_n\}$ be the set of real variables, then a PORV, p, can be represented as, $p ::= f(r_1, r_2, \ldots, r_n) \sim 0$, where f is a mapping $f : \mathbb{R}^n \rightarrow \mathbb{R}$, and $\sim$ is a relational operator from the set $\{\geq, >\}$. In this work, we consider only linear maps for the predicate mapping f. Other relational operators can be derived using $\sim$ and the propositional connectives.

Now we can write the properties of Example 2.2 in the following way using real time temporal operators :

1. $(V_{out} > 3V) \wedge \mathcal{G}_{[0,10\mu]}(reset) \Rightarrow \mathcal{F}_{[0,10\mu]}(V_{out} < 1V)$
2. $(V_{in} > 2V) \wedge \mathcal{G}_{[0,5\mu]}(enable) \Rightarrow \mathcal{F}_{[0,5\mu]}(V_{out} > 1.5V)$

We consider following two examples to elaborate STL further.

Example 2.3 *: If enable is low, v_{out} will be less than 0.2V within $10\mu s$. - This can be expressed by the following safety requirement φ in STL.*

$$\varphi \equiv \neg enable \Rightarrow \mathcal{F}_{[0.10e-6]}(v_{out} < 0.2)$$

Example 2.4 *: If enable is held low for at least $2\mu s$ then v_{out} will become less than 0.2V within $10\mu s$. - We encode this by the following safety requirement φ in STL.*

$$\varphi \equiv \mathcal{G}_{[0.2e-6]}(\neg enable) \Rightarrow \mathcal{F}_{[0.12e-6]}(v_{out} < 0.2)$$

In [51, 52], authors extended STL to derive AMS-LTL which introduces the notion of *event*. In [55], the author discussed the notion of an *event* which requires the past MITL operator *since* to express an event. In AMS-LTL, *event* is expressed in future temporal logic only. STL introduced PORV to allow predicates over signals like voltage, current etc, although many real world behaviors are still not expressible in STL. For example, consider the following requirements.

1. Whenever *enable* goes high, within $10\mu s$, *V(out)* goes above *2V* (a property related to *rise time*).
2. The delay between *V(a)* going above *0.1V* and above *0.85V* is less than the delay between *V(a)* going above *0.85V* and above *0.95V* (a property relating *rise time* and *steady state time*).

Though such requirements are very common while describing timed behaviors of mixed-signal systems, they are certainly not expressible in STL, because they involve events and comparison between time intervals. To be able to express such requirements, the authors extend STL to derive AMS-LTL which introduces the notion of event in the logic formulas.

Definition 2.2.2 [Syntax of AMS-LTL] *: The syntax of AMS-LTL formula* φ *is defined recursively by the following grammar rules.*

$$\varphi ::= \top \mid p \mid E \mid \neg\, \varphi \mid \varphi \,\wedge\, \varphi \mid \varphi\, \mathcal{U}_I\, \varphi$$

E is an event to express change of truth of propositions and is defined by the following grammar rules.

- $E ::= @^{+}(B) \mid @^{-}(B) \mid @(B)$
- $B ::= p \mid \neg B \mid B \wedge B$

A few *safety requirement* properties are expressed below in AMS-LTL :

Example 2.5 *: Whenever* v_a *crosses* $1.3V$ *in the rising direction, within* $0.5\mu s$ *the system should flag failure by asserting c.*
This is a safety requirement which can be expressed in AMS-LTL in following way.

$$\varphi \equiv @^{+}(v_a > 1.3) \Rightarrow \mathcal{F}_{[0,5.0e-7]}(c)$$

Example 2.6 *: If* v_a *crosses* $1.2V$ *in the rising direction, then it should remain above* $1.2V$ *for at least* $20ns$*.*
This is again a safety requirement. The property can be encoded in to AMS-LTL as follows :

$$\varphi \equiv @^{+}(v_a > 1.2) \Rightarrow \mathcal{G}_{[0,2.0e-8]}(v_a > 1.2)$$

Though these recent AMS extensions of temporal logics have a neat semantics which easily follows from the semantics of the underlying temporal logic, the encapsulation of real variables within analog predicates severely constrains the expressibility of AMS behaviors. Researchers have studied the use of auxiliary AMS functions and Auxiliary State Machines with the assertion language [50, 51] to partially address this limitation. For example, consider the following property:

Example 2.7 *After the enable signal becomes high, the output voltage, $V(out)$, will rise following the curve $f(t) = (1 - e^{ct})$ with a tolerance of ϵ for the next $20\mu s$. (f need not to be a simple function of time only, it can have dependencies on other signals and / or can be a complex function of time. For simplicity of understanding, we kept f as a function of time only.)*

In order to monitor this property, the function, $f(t)$, has to be encoded (say, in Verilog-AMS) as an auxiliary function in the simulation framework. This auxiliary model will have to be triggered by the *enable* signal. Let z denotes the output of this auxiliary model (that is, z represents $f(t)$). Now, the required property can be expressed using z as follows:

$$\mathcal{G}(enable \;\Rightarrow\; \mathcal{G}_{[0,20\mu s]}\;(|V(out) - z| < \epsilon)$$

This example shows that for capturing quite simple AMS behaviors, the verification engineer must use a combination of auxiliary functions and the assertion language. This can potentially be quite confusing at times for the verification engineer. Researchers have also studied synchronization of AMS assertions with the AMS simulators in [52] to guarantee that the truth of the assertions are evaluated correctly. The researchers have analyzed impact on the AMS simulator due to the insertion of additional simulation points by the assertion checker to monitor truth of the properties. Insertion of too many additional simulation point may degrade the performance of the simulator considerably. For example, using a sampling clock of fine granularity as advocated in [53] is not a good option in practice and can degrade simulator performance severely.

2.3. Equivalence and Simulation Relations

Equivalence checking is a problem where we are given two system models and are asked whether these systems are equivalent with respect to some notion of conformance or functionally similar w.r.t their input / output behavior. Verification can be based on specific properties like transient or steady state response properties, in time domain or frequency domain. Such conformance relation between designs

is classically done through exhaustive testing by proving that two expressions are equivalent which can be difficult for any large circuit. On the other hand the symbolic reasoning methods can prove or disprove simulation relation or equivalence using the systematic decision procedures over all the valid range of inputs described symbolically. Therefore, it is possible to compare circuits on the same level of abstraction as well as on different levels, e.g. SPICE netlists versus analog behavioral models, behavioral vs. macro model or macro model vs. device level etc.

In the digital domain, formal methods work on logical representations of the the finite state systems. Therefore when we refer to equivalence checking between two digital designs in verification perspective, we essentially mean logical equivalence between the state transition relations of the two. On the other hand the notion of equivalence between two analog circuits is more involved. The most primitive definition of conformance between two analog circuits as appears in [37], is in terms of matching the output of the two models within acceptable limits of tolerance under all input scenarios. Typically, it is a much stringent condition for equivalence in practice because the equivalence may be desired with respect to some features and we may not be interested in which way circuits behave in some other scenarios. Therefore, the definition of formal equivalence checking between Analog-Mixed Signal (AMS) models requires the ability to specify behavioral properties with respect to which we care about equivalence.

Formal Equivalence checking between AMS models has been a subject of considerable research focus in recent times [37, 64]. In [21], the authors proposed a method for applying equivalence checking between two designs (e.g. specification and implementation) of analog systems described by their linear transfer function. The main idea was based on the discretization of the transfer functions to the Z-domain using bilinear transformation and hence the design can be expressed in terms of discrete time components and encoded into Finite State Machine (FSM) representation like Binary Decision Diagram (BDD)s. The verification problem that was attacked in this paper can be stated as follows: "the transient behavior of the implementation mimics that of specification *iff* for any initial state of the specification, there exists a state in the implementation such that the FSMs representing the two circuits produce identical output sequences for all input sequences. The ideas of [21] have been extended in [61] in the following manner. Given the transfer function description of both the specification and the implementation, verify the conformance of the magnitude and the phase response of the implementation against the specification over the desired range of frequency. The equivalence problem in [61] has been modeled as an optimization problem by ensuring the implementation response is bounded within an envelope around

the specification under the variation of the parameter. The conformance [61] is defined using the notion of different frequency bands product response functions (PRFs) of both design models and which serve as objective functions for the global optimization routine. Conformance checking with parameter variations was also investigated in [38], where the authors present an equivalence checking method for linear analog circuits to prove that an actual circuit implementation fulfills a specification in a given frequency interval for all parameter variations. The main idea of the procedure is to compare by inclusion the value sets of transfer functions of specification and implementation. To ensure soundness, the authors choose an over-approximation for the implementation function while an under-approximation is chosen for the specification transfer function. Comparing [21] and [61], we see that in the first work the author trades accuracy for practicality. They adapt BDD based equivalence checking for the analog systems. This comes at the cost of precision which is affected by mainly due to discretization procedure. In contrast, the authors of [61] insist on soundness by checking that the implementation of the behavior is included in the specification behavior.

While the previously mentioned work are concerned with frequency domain behavior, others focus on time domain verification problem. In [39], an equivalence checking approach based on qualitative comparison between two representations of the non-linear analog system is presented. As direct comparison of vector fields for non-linear analog systems is usually not possible, therefore the authors propose to apply non-linear transformations on the sample state spaces to make the comparison possible. Another equivalence checking approach was proposed in [60] for verifying VHDL-AMS designs. The idea consisted of combining equivalence checking, rewriting systems and simulation into a verification environment. The methodology consists of partitioning the specification and implementation codes into digital, analog and data converter components. Digital components are verified using classical equivalence checking whereas analog specification and implementation are simplified using pattern matching. This syntactic method can only be performed on simple designs where rewriting techniques can be easily applied.

In [36], authors have shown that the model checking methods developed for the hybrid dynamic systems may be applied for analog circuit verification. Finite-state abstractions of the continuous analog behavior are automatically constructed using the polyhedral outer approximations to the flows of the underlying continuous differential and difference equations. The authors did not discretize the entire continuous state space and their abstraction captures the relevant behaviors for verification in terms of the transition between states as a finite state machine in the hybrid system model. They have shown their approach on a delta-sigma ($\Delta - \Sigma$) modulator. In [54], researchers have proposed a verification methodology

of analog designs in the presence of process variations and noise with the help of automated theorem proved called MetiTarski [8]. They have proposed the use of stochastic differential equations to model the designs. The proposed approach has been illustrated on an inverting op-amp integrator and a band-gap reference bias circuit. The authors of [31] have proposed two methods to obtain closed form solutions to the model of the circuit - one is based on the piecewise linear modeling and the inverse Laplace transform and the other is based on small-signal analysis and transfer function theory. They transform the properties to be verified into a set of inequalities involving analytic functions and used MetiTarski to prove them automatically. In [35], the authors have proposed a verification approach of the DC and low frequency behavior of synthesized analog designs containing linear components and components whose behaviors can be represented by piecewise linear models. A formal model of the structural description of synthesized design is extracted from the netlist provided by the synthesis tool in terms of characteristic behaviors of the components and the various current and voltage laws. For the synthesized model to be correct it must conform to a formal model extracted from the user given behavior specification. Circuit implementations and the expected behavior are both modeled in PVS [11] higher-order logic proof checker, as linear functions and the PVS decision procedures are used to prove the conformance. In [67], the author has proposed a method where the models of AMS circuits written in VHDL-AMS [12] are converted into *Labeled Hybrid Petri Net* (LHPN) which includes Boolean signals to represent digital circuitry and continuous variables to model voltages and currents in the analog circuitry. The properties to be verified are specified as temporal logic formulae using timed CTL (TCTL) generated from the *assert* statements of VHDL-AMS automatically. These properties are then checked using a LHPN model checker based on SMT solver theory. The authors of [14] have presented an approach to verify locking of charge-pump phase-locked loops by performing reachability analysis on a behavioral model of the circuit. Researchers have considered different piecewise linear (PWL) models for nonlinear devices in the context of formal DC operating point and transient analyses of analog circuits in [69]. PWL models can be encoded as a verification problem as constraints in linear arithmetic which can be solved efficiently using modern SMT solvers. The authors found out the most suitable PWL modeling approach for formal verification by experimentally evaluating the performance of various PWL models in terms of running time and accuracy for the DC operating point and transient analyses of several analog circuits. The authors in [27] have used statistical model checking approach to verify properties of AMS circuits. They have also proposed a logic which can express desired properties in temporal as well as in frequency domain. The authors have demon-

strated the applicability of their method on a third order delta-sigma ($\Delta - \Sigma$) modulator and reported significant gain in terms of the running time compared to previously reported approaches in literature. A novel methodology based on Boolean satisfiability (SAT) has been proposed for formulating a SPICE-type circuit simulation problem as satisfiability problem in [66]. The authors start with a circuit level netlist, capture the non-linear behavior of the circuits at the transistor level via conservative approximations and transform the simulation problem into a search problem that can be exhaustively explored via SAT solver. They have also presented algorithms for abstraction refinement and smart interval generation to improve the computational efficiency of the proposed solution scheme. The ideas have been implemented in a tool called *fSpice*. In [28], the authors have demonstrated an extension of formal verification methodology in order to deal with time-domain properties of analog and mixed-signal circuits whose dynamic behavior is described by differential algebraic equations (DAEs). They have proposed an algorithm for approximating sets of reachable sets for dense-time continuous systems to deal with DAEs and applied it to a biquad low-pass filter. To analyze more complex circuits, they resorted to bounded horizon verification and techniques from optimal control theory. Generic *monitor* automata was proposed in [34] to facilitate the application of hybrid system reachability computations to characterize time domain features of oscillatory behavior, such as bounds on the signal amplitude and jitter. The approach is illustrated for a nonlinear tunnel-diode circuit model using PHAVer [10], a hybrid system analysis tool that provides sound verification results based on linear hybrid automata approximations and infinite precision computations. In [33], the authors have reported a novel approach combining forward and backward reachability while iteratively refining partitions at each step to verify properties of oscillator circuits which needs cyclic invariants. They have also reported a considerable reduction in memory and runtime and illustrated their approach by verifying the limit cycle oscillation behavior of a third-order model of a differential VCO circuit. In [29], a method based upon Rapidly-exploring Random Trees (RRT) (a well known probabilistic path/motion planning technique in robotics with a special property that allows to guarantee a good coverage quality) has been proposed to construct a simulation-based method for validating AMS circuits. The authors investigated conditions for preserving this coverage property and develop a variant of RRTs which is more time-efficient. They have implemented the ideas in a prototype tool that can handle high dimensional hybrid models. The authors in [46] have discussed methodologies for generation of abstract models for AMS circuits and simulation aided verification for the same.

In all the above mentioned works, the methods mostly focus on equivalence

checking between analog circuits but they do not address the equivalence checking of AMS designs with extensive use of digital logic (like battery charger). More recently, researchers have started exploring equivalence checking for large AMS circuits [63], given that the methods for analog equivalence checking do not scale to AMS designs of large size. Another recent work [43] attempts to leverage equivalence checking and coverage analysis methods from the digital domain after discretizing the state space of the analog part of the circuit. This approach is very promising since state space discretization is unavoidable in AMS equivalence checking and being able to leverage existing tools from the digital domain enables the unification of the equivalence checking of both digital and analog components.

An important component of AMS circuits is its digital brain or the digital controller. The digital controller periodically senses the signals of the analog component and actuates control on the basis of the samples it senses. Typically actuation of control takes place at the crossing of some thresholds for one or more real valued signals. For example a Low Dropout Regulator (LDO) goes to short circuit mode when its output current exceeds the short-circuit current threshold. Besides checking the equivalence of analog components, AMS equivalence checking methods must also consider the equivalence between the digital controllers, particularly when they actuate control at possible different thresholds. This work will focus on developing formal methods to check whether the implementation of the digital controller of AMS circuits indeed meets the specification or in other words to find *if there exists a simulation relation between specification and the implementation of the digital controller of the AMS circuits.*.

The problem of finding simulation relation between implementation and specification is essentially a *sequential equivalence checking* [23, 24] problem where unlike combinational equivalence checking, the state mapping between implementation and the specification is not known a priori. There exists two classical *equivalence checking* algorithm in the digital domain namely Kanellakis-Smolka's Algorithm (KS)[45, 44] and Paige-Tarjan Algorithm (PT) [56]. In the next section, we give the formal definition of *simulation relation* in digital domain and then we describe KS Algorithm [45, 44] which will be the basis of our proposed method.

2.4. Algorithm for Equivalence Checking

First we define a few terminologies* which will be used to explain KS algorithm.

Definition 2.4.1 [Transition Systems] *: A transition system TS is a tuple (S, Act, $\rightarrow$, I, AP, L) where,*

* We follow the notion of the book *Principles of Model Checking* by Christel Baier and Joost-Pieter Katoen [20]

- *S is a set of states,*
- *Act is a set of actions,*
- *$\rightarrow \subseteq S \times Act \times S$ is a transition relation,*
- *$I \subseteq S$ is a set of initial states,*
- *AP is a set of atomic propositions, and*
- *$L : S \rightarrow 2^{AP}$ is a labeling function.*

TS is called *finite* if S, Act and AP are finite. ■

Definition 2.4.2 [Partition] *: A* partition *for a set of states S is a set $\Pi = \{B_1, B_2, \ldots, B_k\}$ such that $B_i \neq \emptyset$ (for $0 < i \leq k$), $B_i \cap B_j = \emptyset$ (for $0 < i \leq k$ and $i \neq j$), and $S = \bigcup_{0<i\leq k} B_i$. $B_i \in \Pi$ is called a* block*, $C \subseteq S$ is a* superblock *of Π if $C = B_{i_1} \cup \cdots \cup B_{i_l}$ for some $B_{i_1}, \ldots, B_{i_l} \in \Pi$.* ■

Definition 2.4.3 [Pre Operation] *: Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. For $s \in S$ and $\kappa \in Act$, the set of κ-predecessors of s is defined by*

$$Pre(s, \kappa) = \left\{ s' \in S | s' \xrightarrow{\kappa} s \right\}, Pre(s) = \bigcup_{\kappa \in Act} Pre(s, \kappa).$$

■

Definition 2.4.4 [Post Operation] *: Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system. For $s \in S$ and $\nu \in Act$, the set of ν-successors of s is defined by*

$$Post(s, \nu) = \left\{ s' \in S | s \xrightarrow{\nu} s' \right\}, Post(s) = \bigcup_{\nu \in Act} Post(s, \nu).$$

■

The KS algorithm depends on the concept of *splitter.*

Definition 2.4.5 [Splitter] *: A splitter for a block $B_i \in \Pi$ is a block $B_j \in \Pi$ such that, for some action $a \in Act$, some states in B_i have a-labeled transitions whose target is a state in B_j and others do not. Therefore, if there are some states in B_i which can afford a-labeled transition and others cannot, then we have sufficient reason to distinguish two group of states in B_i and hence it can be split by B_j w.r.t action a in two new blocks :*

$$B_i^1 = \{s \mid s \in B_i \text{ and } s \xrightarrow{a} s', \text{ for some } s' \in B_j\} \text{ and}$$
$$B_i^2 = B_i \setminus B_i^1$$

The splitting results in a refinement Π' of Π as given below:

$$\Pi' = \{B_0,\, B_1,\, \ldots,\, B_i^1,\, B_i^2,\, B_{i+1},\, \ldots,\, B_k\}$$

■

To be precise, C is a *splitter* for Π if there exists a block $B \in \Pi$ with $B \cap Pre(C) \neq \emptyset$ and $B \setminus Pre(C) \neq \emptyset$.

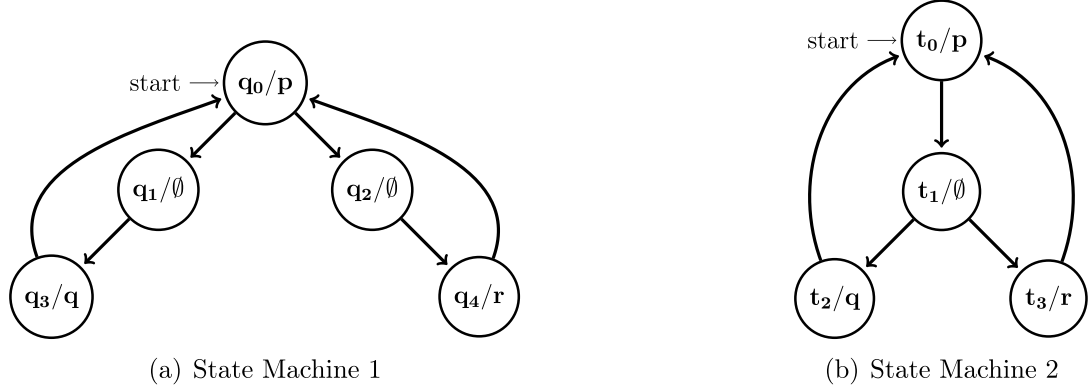


Figure 2.1: TS_2 simulates TS_1

Definition 2.4.6 [Simulation Relation] *: Let TS_i = (S_i, Act_i, $\rightarrow_i$, I_i, AP, L_i), i = 1,2, be the transition system over AP. A simulation for (TS_1, TS_2) is a binary relation $\mathcal{R} \subseteq S_1 \times S_2$ such that*

1. *$\forall s_1 \in I_1$, $\exists s_2 \in I_2$ such that $(s_1, s_2) \in \mathcal{R}$.*
2. *$\forall (s_1, s_2) \in \mathcal{R}$ it holds that*
   - *(a) $L_1(s_1) = L_2(s_2)$.*
   - *(b) if $s_1' \in Post(s_1)$, then there exists $s_2' \in Post(s_2)$ with $(s_1', s_2') \in \mathcal{R}$.*

TS_1 is simulated by TS_2 (denoted by $TS_1 \preceq TS_2$), if there exists a simulation $\mathcal{R}$ for (TS_1, TS_2). Condition 1 of Definition 2.4.6 requires that all initial states of TS_1 are related to at least one initial state of TS_2 but the reverse is not required i.e there can be some initial state in TS_2 that does not have a corresponding candidate state in TS_1. Condition 2a in Definition 2.4.6 states that s_1 and s_2 are equally labeled and hence can be considered "local" equivalence of s_1 and s_2. Condition 2b in Definition 2.4.6 states that every outgoing transition of s_1 must be matched by an outgoing transition of s_2. The reverse of Condition 2b of Definition 2.4.6 may not be true.

In the Figure 2.1 we give an example of *simulation relation*. It is clear that $TS_1 \preceq TS_2$ as every state of TS_1 has a corresponding simulating state in TS_2 but the reverse is not *TRUE* i.e $TS_2 \npreceq TS_1$ since there is no state in TS_1 which can simulate state t_1 of TS_2. Precisely, the *simulation relation* for the two state machines in the Figure 2.1 is as follows :

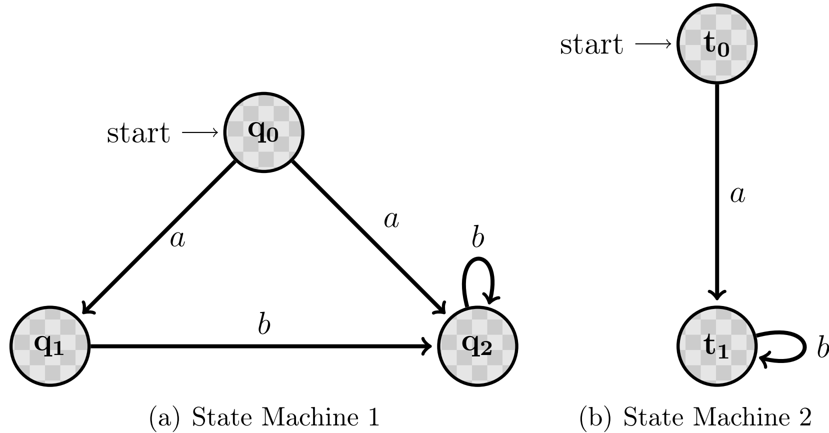
$$\mathcal{R} = \{\{q_0, t_0\},\{q_1, t_1\}, \{q_2, t_1\}, \{q_3, t_2\}, \{q_4, t_3\}\}.$$

Since, $\mathcal{R}$ contains (q_0, t_0), hence indeed $TS_1 \preceq TS_2$.

2.4.1. Kanellakis-Smolka Algorithm

The basic idea of the KS Algorithm is to iterate the splitting of some block B_i by some block B_j w.r.t some action a until no further refinement is possible and hence we have reached a *fixpoint*. The resulting partition is called the *coarsest stable partition*. If the initial states of the two state machines belong to the same partition of the *fixpoint*, then the two state machines are *bi-similar*.

We explain KS Algorithm with the help of the labeled transition systems presented in Figure 2.3[†]



(a) State Machine 1 (b) State Machine 2

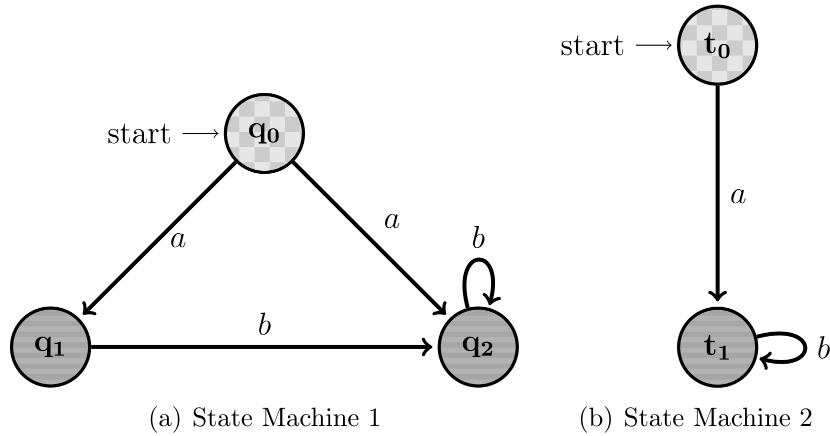
Figure 2.2: Before applying Kanellakis-Smolka's Algorithm

The KS algorithm is meant for checking *bisimilarity* between two state machines. Let the *initial partition* associated with the labeled transition systems of Figure 2.3 be $\{Pr\}$, where

$$Pr = \{q_0,\, q_1,\, q_2,\, t_0,\, t_1\}.$$

The block Pr is a splitter for itself. Indeed some states in Pr can afford a-labeled transitions where other do not. Hence, we can split Pr by Pr based on

[†]This example has been reproduced from *The Algorithmics of Bisimilarity* by L. Aceto, A. Ingolfsdottir and J. Srba published by Cambridge University Press [13]



(a) State Machine 1

(b) State Machine 2

Figure 2.3: After applying Kanellakis-Smolka's Algorithm once

the action a and we get two new partition consisting of the blocks $\{q_0,\ t_0\}$ (the set of states which can afford a labeled transitions) and $\{q_1,\ q_2,\ t_1\}$ (the set of states that cannot afford a labeled transition). We would have obtained the same partition had we done the splitting w.r.t action b. We can observe that neither of the blocks $\{q_0,\ t_0\}$ nor $\{q_1,\ q_2,\ t_1\}$ can be split by the other with respect to any action. Indeed, states in each block are all bisimilar to one another whereas states in different blocks are not. The two state machines are bisimilar since the *initial states* namely q_0 and t_0 of the two state machines are in the same partition i.e. $\{q_0,\ t_0\}$ are in the same partition.

The above mentioned algorithms require that the input labeled transition systems to be fully constructed in advance. In practice, a labeled transition system describing a reactive system is typically specified as a parallel combination of some other labeled transition systems. Hence, it is well known that the size of the resulting labeled transition system may grow exponentially w.r.t the number of the parallel components. The phenomenon is known as *state space explosion* in the verification parlance and hence a major obstacle in the use of the algorithms which require explicit construction of the input labeled transition systems in the automatic verification of large reactive systems.

To deal with the *state space explosion*, a *symbolic approach* based on the use of *Reduced Ordered Binary Decision Diagram* (*ROBDD*) [25] has been proposed in [22] to represent finite labeled transition systems symbolically. The use of ROBDDs permit succinct representation of finite objects.

2.5. Concluding Remarks

This chapter provides a basic overview of the different logic languages of both digital as well as AMS domain. It also describes OVL of digital domain. This

chapter also presents some relevant work on the equivalence checking of AMS circuits and has described a classical algorithm for finding equivalence in the digital domain.

Chapter 3
AMS Verification Library

The development and use of assertions in the Analog and Mixed-signal (AMS) domain is a subject which has attracted significant attention lately from the verification community. Recent studies have suggested that natural extensions of assertion languages (like PSL [7] and SVA [6]) into the AMS domain are not expressive enough to capture many AMS behaviors, and that a library of auxiliary AMS functions [50, 51] are needed along with the assertion language. The integration of auxiliary functions with the core fabric of a temporal logic is non-trivial and can be challenging for a verification engineer. Assertions are widely used in simulation based verification in digital domain and assertion languages like SVA or PSL have their syntax derived from temporal logics like LTL. Research community is now seriously pursuing to extend the assertion languages to AMS domain to capture AMS behaviors. In this chapter, we propose an alternative to using AMS assertions, namely the use of a *verification library*. It is a collection of carefully chosen parameterized basic modules which can be used to perform simple arithmetic, Boolean operation and to model temporal properties on dense real time signal. Complex property verification modules can be composed by connecting these basic modules in a proper way. We derive such a complex property module in Example 3.1 by interconnecting basic modules of AMS Verification Library (AMS-VL). We believe that verification library has some definite advantages in the AMS domain. Some of the prominent advantages are (i) the user need not be well trained in assertion languages and its AMS extensions. The user can easily connect different basic modules to realise complex property verification module, (ii) every module is parameterized and the parameters can be easily set using a GUI in a schematic editor, (iii) verification library and auxiliary function library can be unified to create a common framework for capturing AMS behaviors which is less confusing for verification engineers. Moreover, some behaviors which are easily expressed using the verification library cannot be succinctly expressed using some of the existing AMS extensions of temporal logics like AMS-LTL [51, 52]. We explain one such behavior in Example 3.2 after introducing the basic modules

in the next section.

In our approach, we have used existing standardized resources (like Verilog-AMS) [3] to develop the library of checkers to be used in a passive online monitoring methodology. We demonstrate the use of these library modules in the verification of large real world AMS designs from the power management domain, like Buck regulators, Linear Drop-Out regulators (LDO), and their components. There exists several fundamental differences between our work and the work reported in the only other paper we could locate on AMS verification library, namely [53]. These are as follows:

1. The libraries reported in [53] work on clock boundaries only, which is not suitable for AMS properties. AMS simulators use their own sampling algorithms which typically produce irregular sampling intervals. Our libraries handshake with the AMS simulator and instruct the simulator to improve accuracy near the events by declaring events of interest through *cross events* in Verilog-AMS. Hence in our approach the risk of missing an event is marginalized.

2. The libraries reported in [53] miss a very important point, namely that two independent matches of a property may overlap in time. In order to handle this aspect through a verification library, the library modules must be *reentrant*. In the proposed work, this is achieved by implementing the modules in such a way that they can spawn concurrent threads to handle overlapping matches.

For the second case above, it is important to note that due to the dense time semantics of AMS behaviors, it is possible to have a continuum of matches for a property unless it is triggered by a discrete event like a cross event. This leads to a potential explosion in the number of states for an online monitor, though it is possible to develop offline monitors which work by analyzing real time intervals. Since our goal is to develop a library of *online* monitors, we chose to impose the restriction that all library modules are triggered by events. Note that this was also the case in [53] with the clock event being the trigger.

The chapter is organized as follows. In Section 3.1, we define a few terms and notations that will be used to explain the semantics of the AMS-VL modules. In the same section we define the parameters of the AMS-VL that characterizes different modules. Section 3.2 introduces the AMS-VL and its modules. Section 3.3 represents several example verification network using the modules of AMS-VL. Section 3.4 presents the tool flow and implementation issues. Section 3.5 introduces test cases in brief, the simulation environment and simulation results using industrial test cases. Section 3.6 presents our conclusions.

3.1. Definitions and Preliminaries

First, we present some definitions which will be used later to define semantics of different AMS-VL modules.

Definition 3.1.1 [Time Interval] *: Time Domain can be defined as the ordered collection of all non-negative real numbers* $\mathbb{R}_{\geq 0}$ *and can be denoted as* $\mathbb{T}$. *An interval* $\mathcal{I}$ *is a non-empty convex subset of* $\mathbb{R}_{\geq 0}$ *and can be written as pair over* $\mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0}$ *i.e. the set of non-negative real numbers. An interval can be open, closed or half-open i.e. open at one end (may be left open or right open). An interval can be bounded or unbounded. Every interval* $\mathcal{I}$ *can be expressed as one of the following forms* (a, b), $(a, b]$, $[a, b)$, $[a, b]$, $[a, \infty)$ *or* (a, ∞) *where* $a, b \in \mathbb{R}_{\geq 0}$ *and* $b > a$. *The left and right end points of* $\mathcal{I}$ *are denoted by* $l(\mathcal{I}) = a$ *and* $r(\mathcal{I}) = b$ *respectively. In this interval notation, for* $t, t' \in \mathbb{R}_{\geq 0}$, $(t + \mathcal{I})$ *denotes the interval* $\{t + t' \mid t' \in \mathcal{I}\}$.

Definition 3.1.2 [Minkowski Sum] *: The* ***Minkowski sum****, also known as dilation, of two sets* P_1 *and* P_2 *in euclidean space is defined as* $P_1 \oplus P_2 = \{x_1 + x_2 : x_1 \in P_1, x_2 \in P_2\}$ *i.e. addition of every element of* P_1 *with every element of* P_2. *Of particular interest is the application of this operation to one-dimensional sets consisting of elements of time domain* $\mathbb{T}$. *Precisely,* $\{t\} \oplus [a,b] = [t+a, t+b]$, $[m,n) \oplus [a,b] = [m+a, n+b]$.

Definition 3.1.3 [Signal] *: A finite length signal s over a domain* $\mathbb{D}$ *is partial order function* $s : \mathbb{T} \rightarrow \mathbb{D}$ *whose domain of definition is the interval* $\mathcal{I} = [0, r)$, $r \in \mathbb{Q}_{\geq 0}$. *We denote* $\mid s \mid = r$*, read as the length of the signal s is r. We use the notation* $s[t] = \perp \quad \forall t \geq |s|$.

Definition 3.1.4 [Signal Space] *: Signal space S is the set of continuous timed Analog and Boolean signals, where each continuous timed analog signal* x_a *is a mapping* $x_a : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$*, and each continuous timed Boolean signal* x_b *is a mapping* $x_b : \mathbb{R}_{\geq 0} \rightarrow \{\top, \perp\}$.

The set of Boolean atomic propositions are denoted by $\mathcal{AP}$, whereas the set of continuous variables are denoted by $\mathcal{V}$. Valuations for the elements of $\mathcal{AP}$ over time are Boolean signals, whereas the valuations for elements of $\mathcal{V}$ over time are analog signals.

AMS-VL consists of parametrized modules. Different parameters of the AMS-VL modules make the total parameter set of AMS-VL. Some of the parameters of the paramset are common to several modules. Also some parameters are there

Name of Parameter	*Context of Utilization*
EdgeDirection	*Specifies direction of triggering condition for which analog event is to be generated, +1 or 1 is meant for posedge and -1 is meant for negedge.*
ThresholdValue	*Specifies threshold for comparison of analog values in analog predicate.*
ArithmeticOperator	*Specifies Arithmetic Operation to be performed on the Analog Signals. +1 or 1 is meant for Arithmetic Addition and -1 is meant for Arithmetic Difference.*
ComparisonOperator	*Specifies the Comparison Operator for the Analog Predicate or Analog Condition statement. +1 or 1 is meant for $>$ operation and -1 is meant for $<$ operation.*
BooleanOperator	*Specifies the Boolean Operator for the Boolean Signals. +1 or 1 is meant for logical AND operation and -1 is meant for logical OR operation.*
Delay	*Specifies the time period for which the TRUTH of an expression has to be checked.*
MinimumDelay	*Specifies the minimum time period before which the property checker should not initiate checking of TRUTH of an expression.*
MaximumDelay	*Specifies the maximum time period till which the property checker should keep checking the TRUTH of an expression.*
KeepMatchHighTime	*Specifies the time duration for which the match/fail signal of the property checker module should remain high after the test condition is satisfied/violated.*
TimeTolerance	*Specifies the maximum allowable error on real time scale between the estimated crossing point and the true crossing point.*
ValueTolerance	*Specifies the maximum allowable error on real value scale between the estimated value point and the true value point.*

Table 3.1: Different Parameters and their Context of Utilization

in the paramset which are specifically used for a particular module. The default values of all the parameters of paramset is defined in a header file called *ams_verif_defines.h*. These are the values assumed by the parameters when the user does not specify the values explicitly. User can override default values by passing new values in the instantiation of the module in the schematic. We present name of the different parameters and their context of utilization in the Table 3.1.

3.2. The Structure of AMS Verification Library

The proposed AMS verification library, called AMS-VL, extends the OVL approach to the AMS domain. Like the OVL modules, AMS-VL modules can be interconnected to create composite properties from the atomic properties represented by individual modules. In the next subsection we present the modules available in the library. The AMS-VL library consists of broadly three types of modules, as shown in Table 3.2. Next we discuss all the modules in detail describ-

Type of Module	*Name of Module*	*Purpose of Module*
Latch Modules	*CaptureAndHold*	captures and holds any input digital signal forever until simulation is over.
	GenerateDelay	holds any input digital signal for a specified delay time.
Arithmetic	*ArithmeticOperator*	generates sum or difference of voltage of analog input signals of the modules. With the addition of unit resistors at the input / output ports, sum or difference of currents of the analog input nets can also be calculated.
and	*EventDetector*	monitors analog cross events on an analog input signal with respect to a specified threshold parameter.
Boolean	*EventDetector_Extended*	monitors analog events based on the relative values of their input analog signals.
Operation	*PredicateEvaluator*	compares any analog signal with user specified threshold value.
Modules	*PredicateEvaluator_ Extended*	compares two analog signals based on their relative values.
	BoolOperator	performs standard Boolean operations.
Interval	*GlobalOperator*	checks the truth of an expression over a specified period of time.
Operation	*EventuallyOperator*	checks whether an expression ever becomes true within a specified time frame.
Modules	*UntilOperator*	checks whether an expression remains true over a time window until another event occurs
	PredicateAssert	checks the truth of an expression when a particular condition is satisfied over a specified period of time.

Table 3.2: Broad Classification of AMS-VL Modules

ing their input / output ports, parameters, syntax and semantics.

3.2.1. CaptureAndHold

The module can be used to capture and hold any Boolean / digital signal. For example, this module can capture and hold the match / fail signal of some property, which can be used to trigger the checking of next level property.

Port Name	Port Type	Description
in_1	logic	a Boolean signal is applied in this port.
assertE	logic	signal is asserted when a Boolean signal is captured.

Table 3.3: Ports of CaptureAndHold Module

Parameter Name	Type	Default Value	Range	Unit
N/A	N/A	N/A	N/A	N/A

Table 3.4: Parameters of CaptureAndHold Module

Syntax

CaptureAndHold *instance_name*(in_1, assertE);

Example :

CaptureAndHold CAP_1(in_1, assertE)

3.2.2. GenerateDelay

This module can be used to delay the propagation of any digital pulse. For example, a digital pulse generated after an event is detected, can be propagated through this module after a certain delay value.

Port Name	Port Type	Description
in	logic	any digital pulse that need to be reproduced after certain delay is applied in this port.
out	logic	a pulse gets generated after the delay time corresponding to input digital pulse.

Table 3.5: Ports of GenerateDelay Module

Parameter Name	Type	Default Value	Range	Unit
delay	Real	10e-6	(-inf:inf)	secs

Table 3.6: Parameters of GenerateDelay Module

Syntax

GenerateDelay #(.delay(delay)) *instance_name*(in, out).

Example: The following module GD_1 delays the input Boolean signal *in* by 10μs and propagates it to the output *out*.

GenerateDelay #(.delay(10e-06)) GD_1(in,out).

3.2.3. ArithmeticOperator

Generates voltage equal to the arithmetic sum or arithmetic difference of the input voltages applied at the in_1 and in_2 ports. The module can be used to

add current too. To add current, the user has to add an unit valued resistor in series with the input / output ports of the module. In that case, the input current will be converted to equivalent voltage and the module will add the voltage. The output resistor will convert the output voltage to equivalent current.

Port Name	Port Type	Description
in_1	electrical	A voltage signal is applied at this port.
in_2	electrical	Another voltage signal is applied at this port.
out	electrical	Voltage at this port is arithmetic sum or arithmetic difference of the . input voltage signals applied at the input port in_1 and in_2.

Table 3.7: Ports of ArithmeticOperator Module

Parameter Name	Type	Default Value	Range	Unit
ArithmeticOperator	Integer	1.0	[-1:1] exclude 0	N/A

Table 3.8: Parameters of ArithmeticOperator Module

Syntax

ArithmeticOperator #(.ArithmeticOperator(arithmeticoperator)) *instance_name*(in_1, in_2, out);

Example : The following module AO_1 adds the input voltages at input ports *inp_1* and *inp_2* and produces the sum of voltages at the output port *outp_1*.
ArithmeticOperator #(.ArithmeticOperator(+1)) AO_1(inp_1, inp_2, outp_1)

Description

Whenever the *ArithmeticOperator* is equal to 1 or +1, this module generates the arithmetic sum of the voltages applied at the input ports. Conversely, when the *ArithmeticOperator* is equal to -1, this module generates the arithmetic difference of the voltages applied at the input ports.

3.2.4. EventDetector

Generates a pulse corresponding to an analog event whenever a particular analog condition is SATISFIED.

Port Name	Port Type	Description
in_1	electrical	An analog signal is applied in this port.
match	logic	A pulse of duration equal to KMTH is generated at this port for a monitored analog event when threshold crossing in a specified direction in input analog signal is detected.

Table 3.9: Ports of EventDetector Module

Parameter Name	Type	Default Value	Range	Unit
ThresholdValue	Real	1.0	(-inf:inf)	Volt
ValueTolerance	Real	1e-9	[-1:1]	Volt
TimeTolerance	Real	1e-9	[-1:1]	secs
EdgeDirection	Integer	1	[-1:1]	N/A
KeepMatchHighTime	Real	10e-9	[1e-12:1e-6]	secs

Table 3.10: Parameters of EventDetector Module

Syntax

EventDetector #(.ThresholdValue(valTld), .ValueTolerance(valV), .TimeTolerance(valT), .EdgeDirection(ED), .KeepMatchHighTime(KMHT))
instance_ name(in_ 1, match);

Example : The following module ED_1 detects the following event at the input port *inp_1*: Whenever input voltage exceeds +1.5V an event should be produced. The event pulse duration is 10ns, time tolerance parameters is 1ns, value tolerance parameter is 1μV.

EventDetector #(.ThresholdValue(+1.5), .ValueTolerance(1e-6), .TimeTolerance(1e-9), .EdgeDirection(+1), .KeepMatchHighTime(10e-9)) ED_1(inp_1, match)

We define an analog event b in terms of input signal and parameters of the module as $b = cross(V(in_1) - valTld)$ where *cross* generates monitored analog event b when voltage at *in_1* crosses *valTld*. Here $ED \in \{@^+, @^-\}$. $@^+$ denotes *posedge* and $@^-$ denotes *negedge*. Detection of an analog event by *EventDetetor* is denoted by $\mathcal{ED}$(b). Detection of a *posedge* analog event by *EventDetector* is denoted by $\{\langle S, t\rangle \vdash @^+(b)\} \Rightarrow match_{\uparrow}^{KMTH}$ [*] and detection of a *negedge* analog event is denoted by $\{\langle S, t\rangle \vdash @^-(b)\} \Rightarrow match_{\uparrow}^{KMTH}$ [†]. On detection of an event, a pulse of duration *KeepMatchHighTime* is generated at the match output. The pulse at the match output is generated within a tolerance bounding box defined by the *ValueTolerance* and *TimeTolerance* parameter. Figure 3.1 shows a trace of the module.

[*] read as ⟨ S, t⟩ detects a posedge event b then match is high for KMTH duration
[†] read as ⟨ S, t⟩ detects a negedge event b then match is high for KMTH duration

Semantics

For a signal space S, detection of an analog event b at time t can be expressed as:

$$\mathcal{ED}(b) \Rightarrow \begin{cases} \{\langle S,t\rangle \vdash @^{+}(b)\} \Rightarrow match_{\uparrow}^{KMTH} \begin{cases} \text{if}\langle S,t'\rangle \models b \text{ where } t' \in [t \pm valT] \ \& \\ \exists t'' \in [0,t') \text{ s.t}\forall\ t''' \in [t'',t')\ \langle S,t'''\rangle \not\models b \\ \text{then match} = 1\ \forall t'''' \in [t' + KMHT]\ \& \\ \qquad \text{match} = 0 \text{ elsewhere.} \end{cases} \\ \{\langle S,t\rangle \vdash @^{-}(b)\} \Rightarrow match_{\uparrow}^{KMTH} \begin{cases} \text{if}\langle S,t'\rangle \not\models b \text{ where } t' \in [t \pm valT] \ \& \\ \exists t'' \in [0,t') \text{ s.t}\forall\ t''' \in [t'',t')\ \langle S,t'''\rangle \models b \\ \text{then match} = 1\ \forall t'''' \in [t' + KMHT]\ \& \\ \qquad \text{match} = 0 \text{ elsewhere.} \end{cases} \end{cases}$$

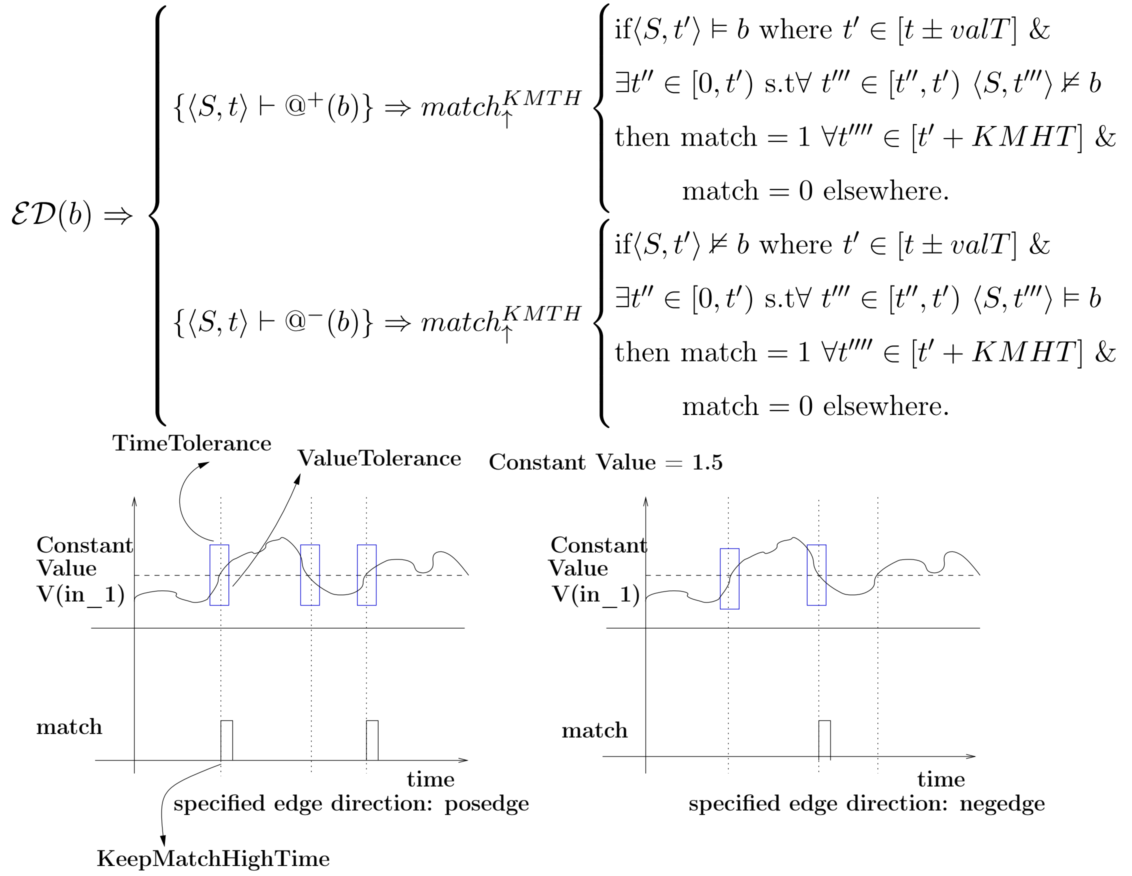


Figure 3.1: Temporal Trace of EventDetector

3.2.5. EventDetector_Extended

Generates a match pulse corresponding to an analog event whenever a particular analog condition is SATISFIED by two analog signals.

Port Name	Port Type	Description
in_1	electrical	One analog signal is applied in this port.
in_2	electrical	Another analog signal is applied in this port.
match	logic	If analog event found, a pulse is generated at match.

Table 3.11: Ports of EventDetector_Extended Module

Syntax

EventDetector_Extended #(.ValueTolerance(valV), .TimeTolerance(valT), .EdgeDirection(ED), .KeepMatchHighTime(KMHT)) *instance_name*(in_1, in_2, match);

Parameter Name	Type	Default Value	Range	Unit
ValueTolerance	Real	1e-9	[-1:1]	Volt
TimeTolerance	Real	1e-9	[-1:1]	secs
EdgeDirection	Integer	1	[-1:1]	N/A
KeepMatchHighTime	Real	10e-9	[1e-12:1e-6]	secs

Table 3.12: Parameters of EventDetector_Extended Module

Example : The following module EDE_1 detetects the following event at the input ports *inp_1* and *inp_2*. Whenever the input signal voltage at *inp_1* exceeds the input voltage at *inp_2*, an event should be produced. The value of value tolernce, time tolerance, pulse width of the match pulse are kept same as in previous subsection.

EventDetector_Extended #(.ValueTolerance(1e-6), .TimeTolerance(1e-9), .EdgeDirection(+1), .KeepMatchHighTime(10e-9)) EDE_1(inp_1, inp_2, match)

We define an analog event b in terms of input and parameters of the module as $b = cross(V(in_1) - V(in_2))$ where cross generates monitored analog event b when voltage at *in_1* crosses voltage at *in_2*. Here $ED \in \{@^+, @^-\}$. $@^+$ denotes *posedge* and $@^-$ denotes *negedge*. Detection of an analog event by *ams_EventDetetor_Extended* is denoted by $\mathcal{EDE}$(b). Detection of a *posedge* analog event by *EventDetector_Extended* is denoted by $\{\langle S, t\rangle \vdash @^+(b)\} \Rightarrow match_{\uparrow}^{KMTH}$ and detection of a *negedge* analog event is denoted by $\{\langle S, t\rangle \vdash @^-(b)\} \Rightarrow match_{\uparrow}^{KMTH}$. On detection of an event, a pulse of duration *KeepMatchHighTime* is generated at the match output. The pulse at the match output is generated within a tolerance bounding box defined by the *ValueTolerance* and *TimeTolerance* parameter. Figure 3.2 shows a trace of the module.

Semantics

The semantics of *EventDetector_Extended* is similar to the semantics of *EventDetector*.

3.2.6. PredicateEvaluator

Checks whether an analog predicate is TRUE and indicates its truth over dense real time continuously until predicate becomes FALSE.

Syntax

PredicateEvaluator #(.ThresholdValue(valTld), .ValueTolerance(valV), .TimeTolerance(valT), .ComparisonOperator(CO)) *instance_name*(in_1, assertE);

Here CO $\in \{>, <\}$. Here, +1 is meant for > and -1 is meant for <. After

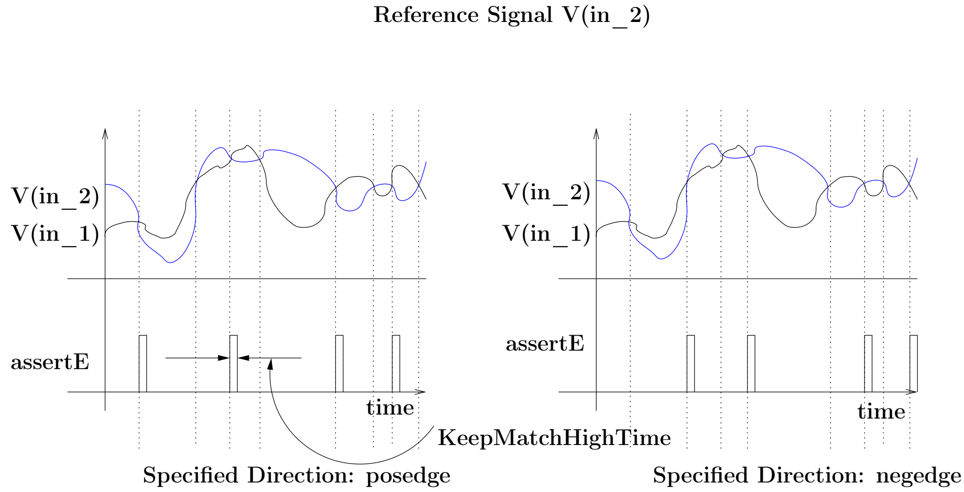


Figure 3.2: Temporal Trace of EventDetector_Extended

Port Name	Port Type	Description
in_1	electrical	An analog signal is applied in this port.
assertE	logic	indicates TRUTH of analog predicate over dense real time continuously by keeping signal at assertE high until predicate becomes FALSE.

Table 3.13: Ports of PredicateEvaluator Module

Parameter Name	Type	Default Value	Range	Unit
ThresholdValue	Real	1	(-inf:inf)	Volt
ValueTolerance	Real	1e-9	[-1:1]	Volt
TimeTolerance	Real	1e-9	[-1:1]	secs
ComparisonOperator	Integer	1	[-1:1] exclude 0	N/A

Table 3.14: Parameters of PredicateEvaluator Module

TRUTH of an analog predicate is detected, signal at port *assertE* will be made high and will be kept high until predicate becomes false.

Example : The following module PE_1 detects whenever the input voltage at *inp_1* becomes greater than 1.5V and keeps assertE high as long as voltage remains higher than 1.5V.

PredicateEvaluator #(.ThresholdValue(+1.5), .ValueTolerance(1e-6), .TimeTolerance(1e-9), .ComparisonOperator(+1)) PE_1(inp_1, assertE)

Ideally analog predicate can be written as:

$$\mathcal{AP}_I = V(in_1) \text{ CO } valTld \text{ where CO} \in \{>, <\}.$$

But in reality it is modeled as follows:

$$\mathcal{AP}_R = V(in_1) \text{ CO } (valTld \pm valV) \text{ where CO} \in \{>, <\}.$$

TRUTH detection of an analog predicate by *PredicateEvaluator* is denoted by $\mathcal{PE}(\mathcal{AP}_R)$. Figure 3.3 shows a trace of the module.

Semantics

For a signal space S, detection of TRUTH of an analog predicate $\mathcal{AP}_R$ over a time interval $\mathcal{I}$ can be expressed as:

$$\mathcal{PE}(\mathcal{AP}_R) = \{\langle S, \mathcal{I}\rangle \vDash \mathcal{AP}_R\} \Rightarrow assertE_{\uparrow} \begin{cases} \text{if } \langle S, l(\mathcal{I})\rangle \vdash @^{+}(\mathcal{AP}_R), \\ \langle S, r(\mathcal{I})\rangle \vdash @^{-}(\mathcal{AP}_R) \text{ and} \\ \forall\, t' \in \mathcal{I}\ \langle S, t'\rangle \vDash \mathcal{AP}_R, \\ \text{then, } assertE \;=\; 1\ \forall t'' \in [\mathcal{I} \pm valT] \\ \qquad\qquad\quad\;\; =\; 0 \text{ elsewhere.} \end{cases}$$

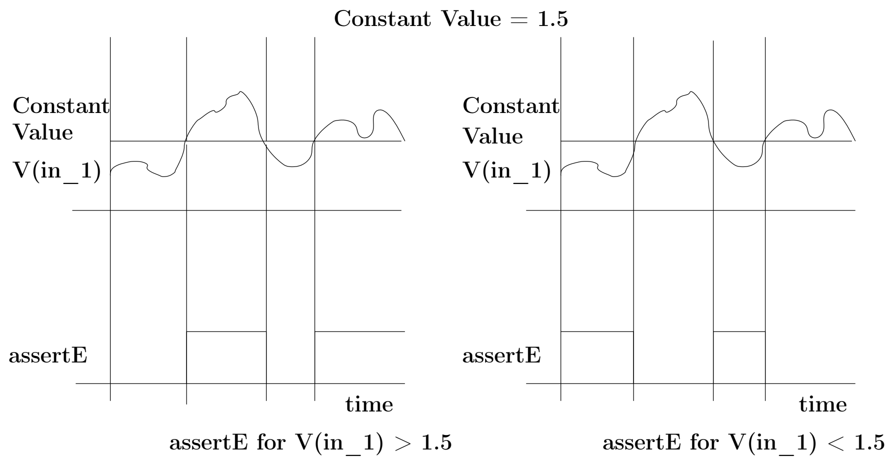


Figure 3.3: Temporal Trace of PredicateEvaluator

3.2.7. PredicateEvaluator_Extended

Checks whether an analog predicate is TRUE and indicates truth over dense real time continuously until predicate becomes FALSE.

Port Name	Port Name	Description
in_1	electrical	An analog signal is applied in this port.
in_2	electrical	Another analog signal is applied in this port.
assertE	logic	indicates TRUTH of analog property over dense real time. continuously by keeping assertE high until property becomes FALSE.

Table 3.15: Ports of PredicateEvaluator_Extended Module

Parameter Name	Type	Default Value	Range	Unit
ValueTolerance	Real	1e-9	[-1:1]	Volt
TimeTolerance	Real	1e-9	[-1:1]	secs
ComparisonOperator	Integer	1	[-1:1] exclude 0	N/A

Table 3.16: Parameters of PredicateEvaluator_Extended Module

Syntax

PredicateEvaluator_Extended #(.ValueTolerance(valV), .TimeTolerance (valT), .ComparisonOperator (CO)) *instance_name*(in_1, in_2, assertE);

Here CO $\in \{>, <\}$. Here, +1 is meant for > and -1 is meant for <. After TRUTH of an analog predicate is detected, signal at port*assertE* will be made high and will be kept high until predicate becomes false.

Example : The following module PED_1 detects whenever the input voltage at *inp_1* becomes greater than input voltage at *inp_2* and keeps assertE high as long as voltage at *inp_1* remains higher than voltage at *inp_2*.

PredicateEvaluator_Extended #(.ValueTolerance(1e-6), .TimeTolerance(1e-9), .ComparisonOperator(+1)) PE_1(inp_1, inp_2, assertE)

Ideally analog predicate can be written as:

$$\mathcal{AP}_I = V(in_1) \text{ CO } V(in_2) \text{ where CO} \in \{>, <\}.$$

But in reality it is modeled as follows:

$$\mathcal{AP}_R = V(in_1) \text{ CO } (V(in_2) \pm valV) \text{ where CO} \in \{>, <\}.$$

TRUTH detection of an analog predicate by *PredicateEvaluator_Extended* is denoted by $\mathcal{PEE}(\mathcal{AP}_R)$.

Semantics

The semantics of *PredicateEvaluator_Extended* is similar to the semantics of *PredicateEvaluator*. An example temporal trace is shown in Figure 3.4.

3.2.8. BoolOperator

Generates Boolean signal by applying Boolean operation on the input Boolean signals.

Syntax

BoolOperator #(.BooleanOperator(BO)) *instance_name*(in_1, in_2, assertE);

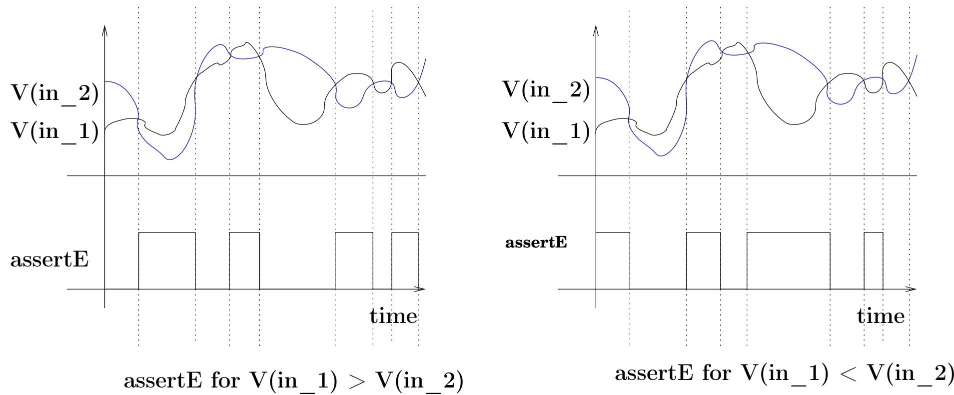


Figure 3.4: Temporal Trace of PredicateEvaluator_Extended

Port Name	Port Type	Description
in_1	logic	A Boolean signal is applied in this port.
in_2	logic	Another Boolean signal is applied in this port.
assertE	logic	After applying Boolean operation on input signals, another Boolean signal. over dense real time is produced at this port.

Table 3.17: Ports of BoolOperator Module

Parameter Name	Type	Default Value	Range	Unit
BoolOperator	Integer	1	[-1:1] exclude 0	N/A

Table 3.18: Parameters of BoolOperator Module

Example : Whenever either of the boolean input (i.e. *inp_1* or *inp_2*) becomes high, the output should become high. It is typically a logical OR operation.

BoolOperator #(.BooleanOperator(-1)) BO_1(inp_1, inp_2, assertE)

Here BO $\in$ {-1, 1} excluding zero. -1 is meant for logical OR operation and +1 or 1 is meant for logical AND operation.

3.2.9. GlobalOperator

After a *start* event occurs, the module checks whether an *expr* remains TRUE for a *delay* period.

Syntax

GlobalOperator #(.TimeTolerance(valT), .delay(delay), .KeepMatchHighTime (KMHT)) *instance_ name*(start, expr, match, fail);

Example: After the *start* signal is received, *expr* should remain high for 20ms.

GlobalOperator #(.TimeTolerance(1e-9), .delay(20e-3), .KeepMatchHighTime(10e-8)) GO_1(start, expr, match, fail)

Port Name	Port Type	Description
start	logic	an event this port triggers checking of *expr*.
expr	logic	expression that is to be checked after start event occurs.
match	logic	if *expr* is TRUE for delay time after *start*, a pulse will be generated in this port to indicate SATISFACTION.
fail	logic	if *expr* is FALSE after *start* event occurs or *expr* become false before delay expires, a pulse is generated at this port immediately to indicate FAILURE.

Table 3.19: Ports of GlobalOperator Module

Parameter Name	Type	Default Value	Range	Unit
TimeTolerance	Real	1e-9	[-1:1]	secs
delay	Real	10000	[0:inf)	ns
KeepMatchHighTime	Real	10e-9	[1e-12:1e-6]	secs

Table 3.20: Parameters of GlobalOperator Module

Truth checking of an *expr* by *GlobalOperator* is denoted by $\mathcal{GO}_{[delay]}(start, expr)$. Satisfaction of an *expr* by *GlobalOperator* is denoted by $\mathcal{GO}^{S}_{[delay]}$ and failure is denoted by $\mathcal{GO}^{F}_{[delay]}$. Algorithm 1 shoes the algorithm and Figure 3.5 shows a trace of *GlobalOperator*.

Semantics

For a signal space S, TRUTH checking of an expression *expr* by *GlobalOperator* at time t can be expressed as $\mathcal{GO}_{[delay]}(start,\ expr) = \varphi^{\mathcal{G}}$:

$$\varphi^{\mathcal{G}} \begin{cases} \{\langle S,t\rangle \vDash \mathcal{GO}^{S}_{[delay]}\} \Rightarrow match^{KMHT}_{\uparrow} \begin{cases} \textit{if } \exists t' < t \ \&\ t \in \mathcal{I} = t' \oplus [delay - valT, \\ delay\ +\ valT], \langle S,t'\rangle \vdash @^{+}(start)\ \& \\ \forall t'' \in [t',t]\langle S,t''\rangle \vDash expr \\ \text{then match} = 1\ \forall t''' \in [t + KMHT]\ \& \\ \qquad \text{match} = 0 \text{ elsewhere.} \end{cases} \\ \\ \{\langle S,t\rangle \vDash \mathcal{GO}^{F}_{[delay]}\} \Rightarrow fail^{KMHT}_{\uparrow} \begin{cases} \text{if } \exists t' < t \ \&\ t \in [t', t' \oplus [delay - valT, \\ delay + valT]),\ \langle S,t'\rangle \vdash @^{+}(start),\ \text{and} \\ \textbf{either } \langle S,t'\rangle \nvDash expr \\ \text{then fail} = 1\ \forall t'' \in [t' + KMHT]\ \& \\ \qquad \text{fail} = 0 \text{ elsewhere.} \\ \\ \textbf{or } \forall t'''\langle S,t'''\rangle \vDash expr \text{ where } t' < t''' \\ < [t' + delay]\ \&\ \langle S,t'''\rangle \vdash @^{-}(expr) \\ \text{then fail} = 1\ \forall t'''' \in [t''' + KMHT]\ \& \\ \qquad \text{fail} = 0 \text{ elsewhere.} \end{cases} \end{cases}$$

Algorithm 1 Algorithm for GlobalOperator Module

1: wait for a start
2: **repeat**
3: **for** every start pulse received **do**
4: **parbegin**
5: initiate a copy of $RTCV$ at delay value.
6: decrease $RTCV$ continuously and monitor *expr*.
7: **if** *expr* is high from initialization of $RTCV$ until it is zero **then**.
8: make *match* high immediately for *short duration* after $RTCV$ is zero.
9: **else if** *expr* is low after initialization of RTCV **or** *expr* gets low before delay time **then**
10: keep *match* de-asserted.
11: make *fail* high immediately for *short duration*.
12: **end if**
13: flush that copy of $RTCV$.
14: **parend**
15: **end for**
16: **until** no start event occurs

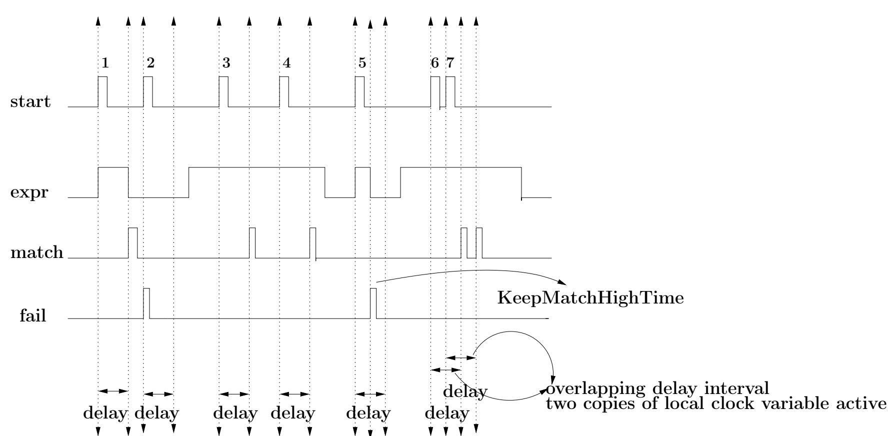


Figure 3.5: Temporal Trace of GlobalOperator

In the Figure 3.5, after the *start pulse* 1 is received, the *expr* signal is continuously checked for the *delay* amount of time. Since *expr* remains high for the entire delay period, hence at the end of the *delay* period, a *match* signal is produced. But for the *start pulse* 2, the *expr* remains low and hence immediately a *fail* signal is produced. Further we note the case for *start pulse* 6 and 7. Their delay period is overlapped. Hence, one parallel thread for each of the *start pulse* 6 and 7 is generated to check *expr* signal independently for the *delay* time. Since in this case *expr* remains high for the delay period for both the *start* pulses, hence two consecutive *match* signals are produced corresponding to the two *delay* periods.

3.2.10. EventuallyOperator

After a *start* event occurs, the module checks whether an *expr* become TRUE within a time window specified by *minimum delay* and *maximum delay.*

Port Name	Port Type	Description
start	logic	an event in this port triggers checking of expr.
expr	logic	expression that is to be checked after start event occurs.
match	logic	if expr becomes TRUE at least once between MinimumDelay and MaximumDelay after start event, then a pulse is generated in this port to indicate SATISFACTION immediately.
fail	logic	if expr never becomes TRUE within the time window specified by MinimumDelay and MaximumDelay time after start event occurs, then a pulse is generated at this port to indicate failure.

Table 3.21: Ports of EventuallyOperator Module

Parameter Name	Type	Default Value	Range	Unit
TimeTolerance	Real	1e-9	[-1:1]	secs
MinimumDelay	Real	0	[0:inf)	ns
MaximumDelay	Real	10000	[0:inf)	ns
KeepMatchHighTime	Real	10e-9	[1e-12:1e-6]	secs

Table 3.22: Parameters of EventuallyOperator Module

Syntax

EventuallyOperator #(.TimeTolerance(valT), .MinimumDelay(MinD), .MaximumDelay(MaxD), .KeepMatchHighTime(KMHT)) *instance_ name*(start, expr, match, fail);

Example : After the *start* signal is received, *expr* should become high within 10ms and 20ms.

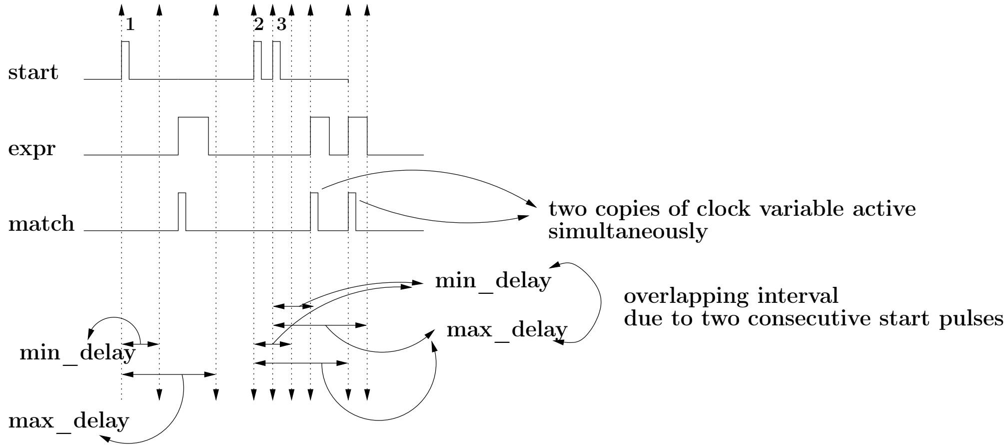


Figure 3.6: Temporal Trace of EventuallyOperator

EventuallyOperator #(.TimeTolerance(1e-9), ,MinimumDelay(10e-3), .MaximumDelay(20e-3), .KeepMatchHighTime(10e-8)) EO_1(start, expr, match, fail)

Truth checking of an *expr* by *EventuallyOperator* is given by $\mathcal{ED}_{[MinD,\ MaxD]}(start, expr)$. Satisfaction of an *expr* by *EventuallyOperator* is denoted by $\mathcal{EO}^{S}_{[MinD,\ MaxD]}$ and failure is denoted by $\mathcal{EO}^{F}_{[MinD,\ MaxD]}$. Algorithm 2 presents the algorithm and Figure 3.6 shows a trace of the operator module.

Semantics

For a signal space S, TRUTH checking of an expression *expr* at time t by *EventuallyOperator* can be expressed as $\mathcal{EO}_{[MinD,\ MaxD]}(start,\ expr) = \varphi^{\mathcal{E}}$:

$$\varphi^{\mathcal{E}} \begin{cases} \{\langle S,t\rangle \vDash \mathcal{EO}^{S}_{[MinD,\ MaxD]}\} \Rightarrow match^{KMHT}_{\uparrow} \begin{cases} \text{if } \exists t' < t,\ t \in \mathcal{I} = t' \oplus [MinD \pm valT, \\ MaxD \pm valT],\ \langle S,t'\rangle \vdash @^{+}(start), \text{and,} \\ \textbf{either } \langle S, l(\mathcal{I})\rangle \vDash expr \\ \text{then match} = 1\ \forall t'' \in [l(\mathcal{I}) + KMHT]\ \& \\ \qquad \text{match} = 0 \text{ elsewhere.} \\ \\ \textbf{or } \exists t''' \in \mathcal{I}, \langle S,t'''\rangle \vDash expr \\ \text{then match} = 1\ \forall t'''' \in [t''' + KMHT]\ \& \\ \qquad \text{match} = 0 \text{ elsewhere.} \end{cases} \\ \\ \{\langle S,t\rangle \vDash \mathcal{EO}^{F}_{[MinD,\ MaxD]}\} \Rightarrow fail^{KMHT}_{\uparrow} \begin{cases} \text{if } \exists t' < t\ \& t \in \mathcal{I} = t' \oplus [MinD \pm valT, \\ MaxD \pm valT],\ \langle S,t'\rangle \vdash @^{+}(start), \\ \&\ \forall t'' \in \mathcal{I},\ \langle S,t''\rangle \nvDash expr \\ \text{then fail} = 1\ \forall t''' \in [r(\mathcal{I}) + KMHT]\ \& \\ \qquad \text{fail} = 0 \text{ elsewhere.} \end{cases} \end{cases}$$

In the Figure 3.6, after the *start pulse* 1 is received, the *expr* signal is not monitored until *min_ delay* time is over. After that the *expr* signal is monitored until *max_ delay* time. Since *expr* became high in between *min_ delay* and *max_ delay*

Algorithm 2 Algorithm for EventuallyOperator Module

```
1: wait for a start pulse
2: repeat
3:     for every start pulse received do
4:         parbegin
5:         initiate a copy of RTCV initialized at real time 0.
6:         start incrementing the RTCV as simulation time progresses.
7:         keep waiting and ignore expr until RTCV exceeds min_delay.
8:         while min_delay ≤ RTCV ≤ max_delay do
9:             keep checking expr.
10:            if expr is high at least once then
11:                make match high for a short duration
12:                keep fail low.
13:                flush that copy of RTCV immediately.
14:            else if expr is never high then
15:                make fail high for a short duration.
16:                keep match low.
17:                flush that copy of RTCV immediately.
18:            end if
19:        end while
20:        parend
21:    end for
22: until no start event occurs
```

hence a *match* signal is produced as soon as the *expr* became asserted. For *start pulse* 2 and 3, two parallel threads of checking *expr* are created corresponding to each of the *start* pulse 2 and 3. As in the both the cases, *expr* becomes high within the stipulated *min_ delay* and *max_ delay* time, hence, two *match* pulses are produced.

3.2.11. UntilOperator

After a *start* event occurs, the module checks whether an *expr* is TRUE in a time window specified by *minimum delay* and *maximum delay*, until a *reset* event occurs.

Syntax

UntilOperator #(.TimeTolerance(valT), .MinimumDelay(MinD), .MaximumDelay(MaxD), .KeepMatchHighTime(KMHT)) *instance_name*(start, expr, reset, match, fail);

Truth checking of an *expr* by *UntilOperator* is denoted by $\mathcal{UO}_{[MinD,\ MaxD]}$ $(start, expr, reset)$. Satisfaction of an *expr* by *UntilOperator* is denoted by $\mathcal{UO}^{S}_{[MinD,\ MaxD]}$ and failure is denoted by $\mathcal{UO}^{F}_{[MinD,\ MaxD]}$. Algorithm 3 shows the algorithm and Figure 3.7 shows an example trace of the module.

Port Name	Port Type	Description
start	logic	an event in this port triggers checking of expr.
expr	logic	expression that is to be checked after start event occurs.
reset	logic	an event in this port ends checking of expr.
match	logic	if expr remains TRUE from MinimumDelay time until an event occurs at reset port, then a pulse is generated in this port to indicate SATISFACTION.
fail	logic	if expr is never TRUE within the time frame or if reset never occurs within time frame although expr is true, then a pulse is generated at this port to indicate FAILURE.

Table 3.23: Ports of UntilOperator Module

Parameter Name	Type	Default Value	Range	Unit
TimeTolerance	Real	1e-9	[-1:1]	secs
MinimumDelay	Real	0	[0:inf)	ns
MaximumDelay	Real	10000	[0:inf)	ns
KeepMatchHighTime	Real	10e-9	[1e-12:1e-6]	secs

Table 3.24: Parameters of UntilOperator Module

Semantics

For a signal space S, TRUTH checking of an expression *expr* at time t by *UntilOperator* can be expressed as $\mathcal{UO}_{[MinD,\ MaxD]}(start,\ expr,\ reset) = \varphi^{\mathcal{U}}$:

$$
\varphi^{\mathcal{U}}
\begin{cases}
\{\langle S,t\rangle \vDash \mathcal{UO}^{S}_{[MinD,\ MaxD]}\} \Rightarrow match^{KMHT}_{\uparrow}
\begin{cases}
\text{if } \exists t' < t \ \&\ t \in \mathcal{I} = t' \oplus [MinD \pm valT, \\
MaxD \pm valT], \langle S,t'\rangle \vdash @^{+}(start), \\
\&\ \exists t'' \in \mathcal{I}, \langle S,t''\rangle \vdash @^{+}(reset) \\
and\ \forall t''' \in [l(\mathcal{I}), t''] \\
\langle S,t'''\rangle \vDash expr, \\
\text{then match} = 1\ \forall t'''' \in [t'' + KMHT]\ \& \\
\qquad \text{match} = 0 \text{ elsewhere.}
\end{cases}
\\
\{\langle S,t\rangle \vDash \mathcal{UO}^{F}_{[MinD,\ MaxD]}\} \Rightarrow fail^{KMHT}_{\uparrow}
\begin{cases}
\text{if } \exists t' < t \ \&\ t \in \mathcal{I} = t' \oplus [MinD \pm valT, \\
MaxD \pm valT], \langle S,t'\rangle \vdash @^{+}(start), \\
\textbf{either},\ \exists t'' \in \mathcal{I}, \langle S,t''\rangle \vdash @^{+}(reset) \\
\text{but } \forall t''' \in [l(\mathcal{I}), t''], \langle S,t'''\rangle \nvDash expr, \\
\text{then fail} = 1\ \forall t'''' \in [t'' + KMHT]\ \& \\
\qquad \text{fail} = 0 \text{ elsewhere.} \\
\\
\textbf{or } \forall t''''' \in [l(\mathcal{I}), r(\mathcal{I})]\ \langle S,t'''''\rangle \vDash expr \\
\text{and } \ \nexists t'''''' \in [l(\mathcal{I}), r(\mathcal{I})] \text{ such that} \\
\langle S,t''''''\rangle \vdash @^{+}(reset) \\
\text{then fail} = 1\ \forall t'''''' \in [r(\mathcal{I}) + KMHT]\ \& \\
\qquad \text{fail} = 0 \text{ elsewhere.}
\end{cases}
\end{cases}
$$

In Figure 3.7, after the *start pulse* 1 is received, *expr* is not monitored until

Algorithm 3 Algorithm for UntilOperator Module

1: wait for a start pulse
2: **repeat**
3:     **for** every start pulse received **do**
4:         **parbegin**
5:         initiate a copy of $RTCV$ initialized at real time 0.
6:         start incrementing the $RTCV$ as simulation time progresses.
7:         keep waiting and ignore *expr* and *reset* until $RTCV$ exceeds *min_ delay.*
8:         **while** $min_delay \leq RTCV \leq max_delay$ **do**
9:             keep monitoring *expr* and *reset.*
10:             **if** *expr* is high continuously until *reset* is asserted **then**
11:                 make *match* high for short duration as soon as *reset* is asserted.
12:                 keep *fail* low.
13:                 flush that copy of $RTCV$ immediately.
14:             **else if** *reset* is not asserted but *expr* remains asserted **then**
15:                 make *fail* high for short duration as soon as *MaximumDelay* period expires.
16:                 keep *match* low.
17:                 flush that copy of $RTCV$ immediately.
18:             **else if** *expr* becomes low before *reset* occurs **then**
19:                 make *fail* high for short duration as soon as *MaximumDelay* period expires.
20:                 keep *match* low.
21:                 flush that copy of $RTCV$ immediately.
22:             **end if**
23:         **end while**
24:         **parend**
25:     **end for**
26: **until** no start event occurs

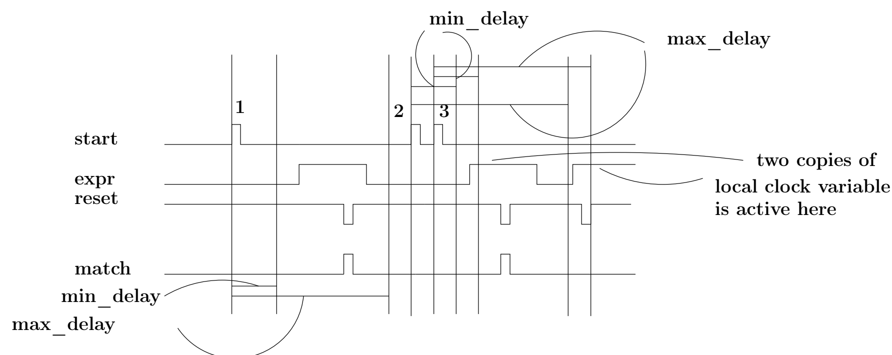


Figure 3.7: Temporal Trace of UntilOperator

min_ delay time. After that the *expr* is monitored upto *max_ delay* time. In this case *expr* becomes high within that time and also the *reset* signal occurs within *max_ delay* time. Hence, a *match* pulse is produced as soon as *reset* pulse occurs. If either *reset* does not occur within the *min_ delay* and *max_ delay* time or *expr* remains low in the time span from *min_ delay* to *max_ delay* or *expr* becomes low

before *reset* occur, then a *fail* pulse should be asserted.

3.2.12. PredicateAssert

After *start* is enabled and kept asserted, it checks whether the *expr* remains TRUE over a period of time. This module is helpful when we want to check a particular property at a particular mode of operation of AMS circuit. For example, we may be interested in checking the variation of steady state voltage of a circuit whenever the circuit is in the steady state mode. We can apply the steady state mode signal of the circuit in the *start* port of *PredicateAssert* and can put the voltage variation as at the *expr* of the module. As long as the circuit is in the particular mode satisfying the desired condition, the *assertE* will remain high.

Port Name	Port Name	Description
start	logic	initiates checking of expr after signal at this port gets asserted.
expr	logic	expression that is to be checked after start asserted.
assertE	logic	signals at assertE will be kept high following some rules described in algorithm as TRUTH condition persists.
failE	logic	signals at failE will be kept high following some rules described in algorithm as TRUTH condition does not persist.

Table 3.25: Ports of PredicateAssert Module

Parameter Name	Type	Default Value	Range	Unit
TimeTolerance	Real	1e-9	[-1:1]	secs
delay	Real	10e-6	[0:inf)	secs

Table 3.26: Parameters of PredicateAssert Module

Syntax

PredicateAssert #(.delay(D), .TimeTolerance(valT)) *instance_name*(start, expr, assertE, failE);

Truth checking of an *expr* by *PredicateAssert* is denoted by $\mathcal{PA}_{[delay]}(start, expr)$. Satisfaction of an *expr* by *PredicateAssert* is denoted by $\mathcal{PA}^{S}_{[delay]}$ and failure is denoted by $\mathcal{PA}^{F}_{[delay]}$. Algorithm 4 presents the module's working principle and Figure 3.8.

Description

This modules contains four pins namely *start*, *expr*, *assertE* and *failE*. Until *start* is asserted, we ignore *expr*. As soon as *start* is asserted, we initiate a local copy of *RTCV*, initialized at the *delay* value (to be supplied by the user). As the simulation time progresses, *RTCV* is decreased continuously. If *expr* remains asserted then *assertE* is made high at the end of delay value and is kept high continuously

according to the following rules:

Rule 1:
If *expr* is high for at least *delay* value after *start* is de-asserted, keep *assertE* high further equal to *delay* value, after *start* is de-asserted and after that assert *failE*. Else, if *expr* gets de-asserted in less than *delay* time after *start* is de-asserted, de-assert *assertE* immediately assert *failE*.

Rule 2:
If *expr* is de-asserted before *start* gets de-asserted, de-assert *assertE* immediately and assert *failE*.

Rule 3:
If *start* and *expr* are de-asserted simultaneously, then de-assert *assertE* immediately and assert *failE*.

The trace over a signal for the above mentioned case is shown below.

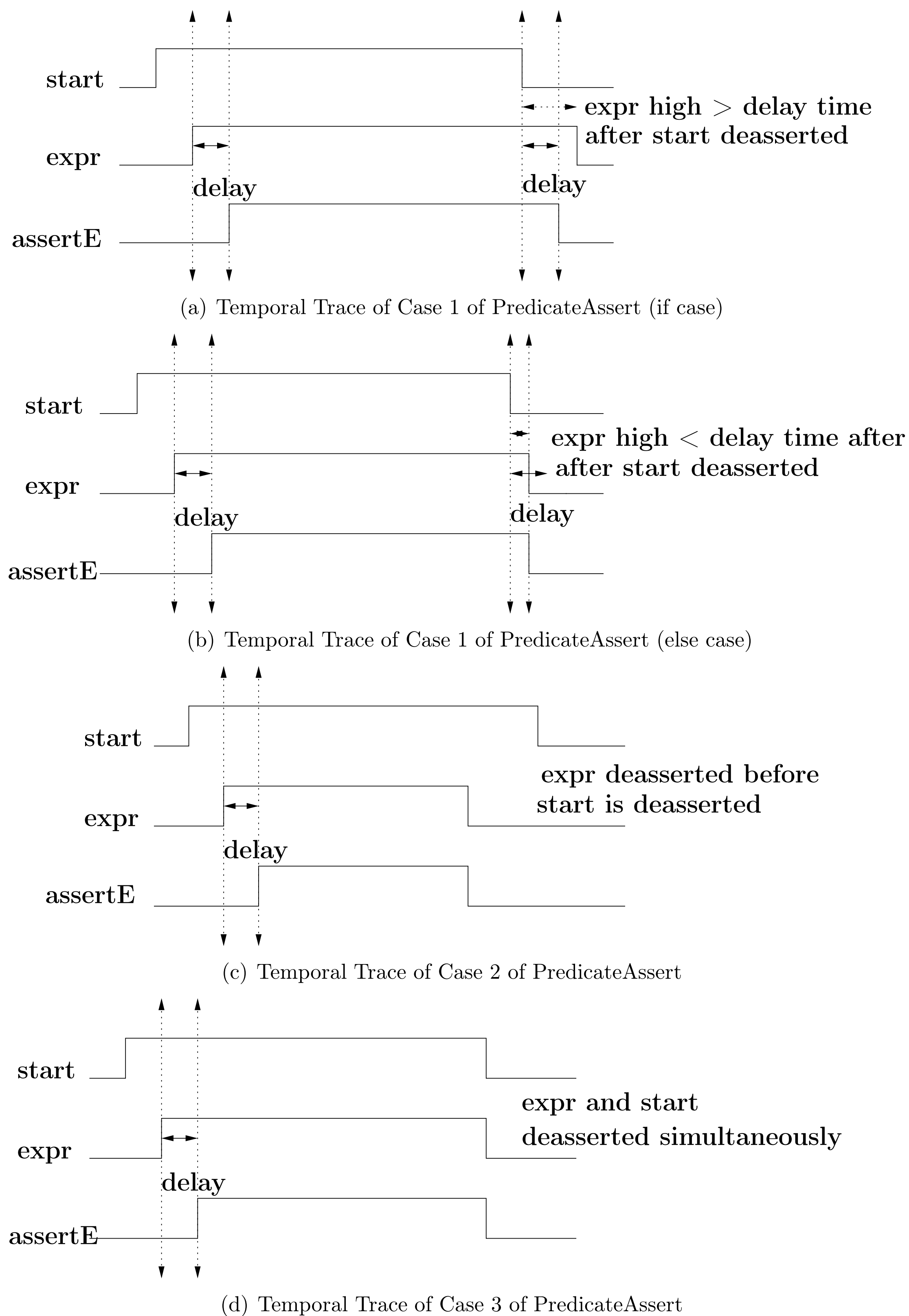
In the next section, we present a few examples to highlight the use these modules.

3.3. Representative Verification Networks with AMS-VL Components

Example 3.1 *Derive a module which can monitor crossing of 3V of an analog signal with a deglitch period of 10μs.*

We use three basic modules namely EventDetector, PredicateEvaluator and GlobalOperator. The EventDetector generates an analog event whenever V_{in} crosses 3V in positive direction (indicated by $+1$) and this match pulse in turn excites the GlobalOperator module (the *start* pulse). The PredicateEvaluator keeps its assertE high as long as V_{in} remains above 3V. GlobalOperator checks for 10μs after receiving *start* pulse whether *expr is high continuously* (i.e. $V_{in} > 3$ for 10μs continuously). If *expr* remains high, then at the end of 10μs, a match pulse will be produced indicating that indeed a deglitched event has occurred. The module has been shown in Figure 3.9. ■

Example 3.2 *: If events p, q and r occur in any order, then event t will occur subsequently.*



(a) Temporal Trace of Case 1 of PredicateAssert (if case)

(b) Temporal Trace of Case 1 of PredicateAssert (else case)

(c) Temporal Trace of Case 2 of PredicateAssert

(d) Temporal Trace of Case 3 of PredicateAssert

Figure 3.8: Different Cases of PredicateAssert Module

Algorithm 4 Algorithm for PredicateAssert Module

1: **repeat**
2: wait for *start*.
3: **if** *start* is asserted **then**
4: start monitoring *expr*.
5: **else**
6: ignore *expr*.
7: **end if**
8: **if** *start* is high and *expr* gets asserted **then**
9: initiate a copy of $RTCV$ immediately initialized at *delay*..
10: **end if**
11: **if** *start* does not get de-asserted and *expr* is asserted until $RTCV$ is zero **then**
12: assert *assertE* and *failE* high as per following:
13: **Case 1:**
14: **if** *expr* is *high* for at least *delay* time after *start* is de-asserted **then**
15: make *assertE* high a maximum of delay time units after *start* is de-asserted.
16: make *failE* high after delay period expires.
17: **else if** *expr* is high less than *delay* time after *start* de-asserted **then**
18: de-assert *assertE* as soon as *expr* is de-asserted and assert *failE*.
19: **end if**
20: **Case 2:**
21: **if** *expr* is de-asserted but *start* continues to remain asserted **then**
22: de-assert *assertE* as soon as *expr* is de-asserted and assert *failE*.
23: **end if**
24: **Case 3:**
25: **if** *expr* and *start* get de-asserted simultaneously **then**
26: de-assert *assertE* as soon as *expr* is de-asserted and assert *failE*.
27: **end if**
28: **end if**
29: **until** false

It is easy to express that p, q and r occurs in any sequence as $\mathcal{F}p \wedge \mathcal{F}q \wedge \mathcal{F}r$ in LTL, and also easy to express that t will follow any of these events (say p) as:

$$\mathcal{G}(p \;\Rightarrow\; \mathcal{F}t)$$

but it is not easy to express that all three events will be followed by t. For example, the property:

$$\mathcal{G}(\; \mathcal{F}p \wedge \mathcal{F}q \wedge \mathcal{F}r \;\Rightarrow\; \mathcal{F}t)$$

does not capture the desired intent, since it does not force event t to occur *after* events p, q and r. In order to express the desired intent in LTL, we have to

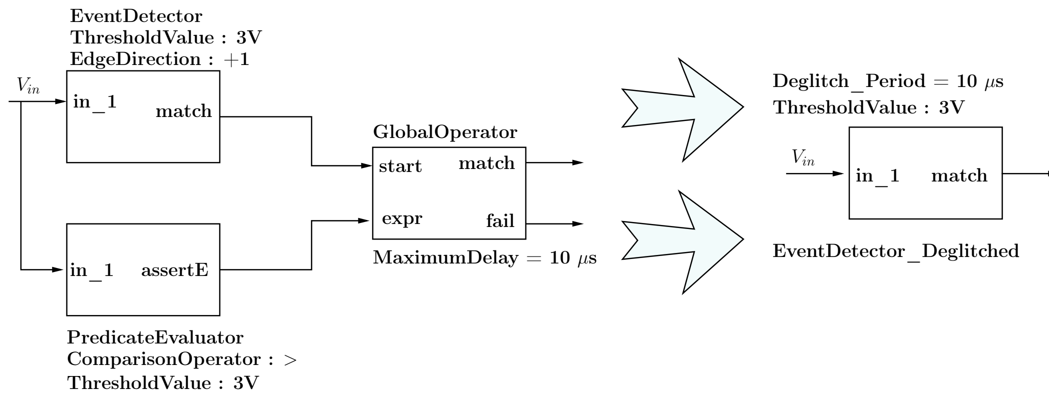


Figure 3.9: EventDetector Deglitched Module

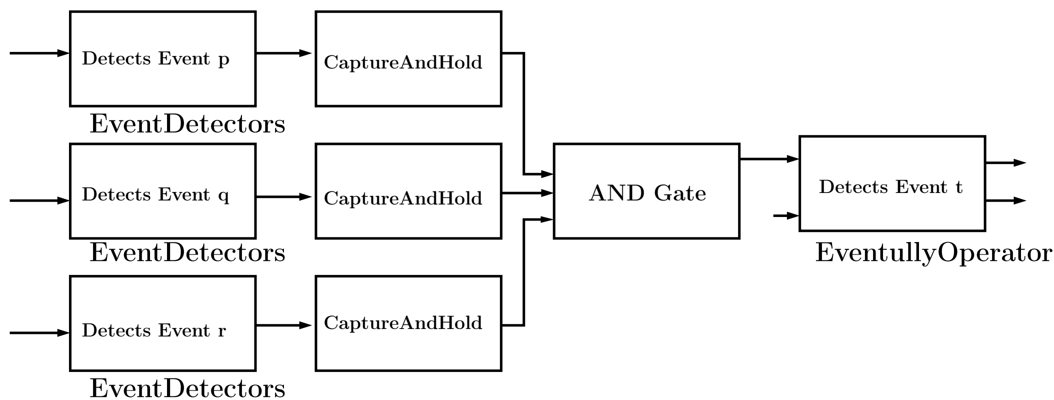


Figure 3.10: Multiple Event Detection

enumerate all six possible sequences in which p, q and r occur and then for each sequence expression that it will be followed by t. Figure 3.10 shows a monitor for the desired property using modules of verification library. The *EventDetectors* detects occurrence of events *p, q and r* in any order and they get latched in the *CaptureAndHold* modules. The output of the *CaptureAndHold* modules are logically ANDed, which triggers checking of event t. The option of using state elements such as latches (i.e. *CaptureAndHold*) and combinational elements such as gates along with the monitors provided in the proposed verification library is a significant advantage over pure formal properties. ■

Example 3.3 : *After v_{in} crosses 2.0 volts, v_{out} should cross 2.5 volts sometime between 10μs and 20μs.*

Figure 3.11 shows the AMS-VL realization of the Property 3.3. In Figure 3.11, the *EventDetector* module monitors the event of v_{in} crossing 2.0 volts. The *match* pulse of this module triggers the *EventuallyOperator* module. The *PredicateEvaluator* compares the voltage at v_{out} and keeps its *assertE* pin high as long as v_{out}

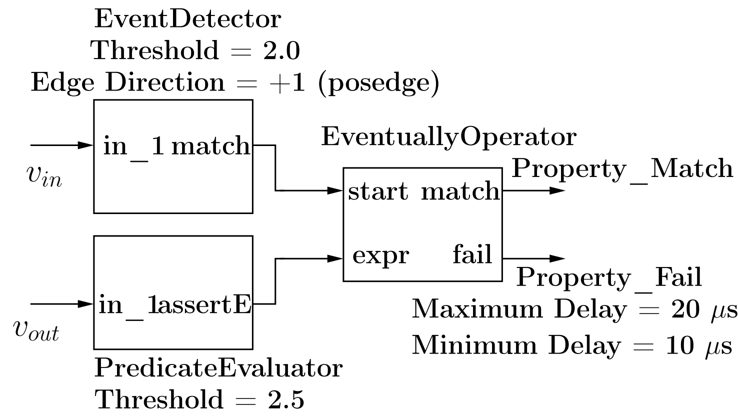


Figure 3.11: AMS-VL Realization of Example 3.3

remains above 2.5 volts. The *EventuallyOperator* module has parameters *MinimumDelay* and *MaximumDelay* which, in this case, are set to 10μs and 20μs respectively. If *assertE* goes high at some time between 10μs and 20μs of receiving its *start* pulse, the *EventuallyOperator* will assert its *Property_ Match* pin, thereby signaling a *match*. On the other hand, if *assertE* does not get asserted within the specified time bound, then *Property_ Fail* will be asserted at the end of 20 μs (that is, after *MaximumDelay*) to indicate a property failure. ■

We now present a more complex example in which the network of AMS-VL modules uses a feedback loop to express a recurring property expression. The following property has been checked on BUCK regulator from National Semiconductor.

Example 3.4 : *After 190 μs of entering startup mode, the buck regulator will enter its steady state mode, where its steady state voltage will remain within 0.5 V with a tolerance of 0.05 V for the next 10 μs. The steady state voltage should remain in this range at a sampling granularity of 20 μs.*

Figure 3.12 shows the AMS-VL realization of the Property 3.4.In Figure 3.12, *BUCK_ EN* is the enable pin of the buck regulator, and *BUCK_ FB* is a feedback pin of the BUCK regulator used to implement our property. In Part A of Figure 3.12, the network detects whether the buck regulator has entered its startup mode. In Part B of the same figure, the network detects whether the buck regulator has entered its steady state mode. In Part C of the figure, the *GenerateDelay* operator activates the *GlobalOperator* module 190μs after start up is detected (by Part A). Within the next 10μs the *GlobalOperator* module will either assert its *match* signal or its *fail* signal, following which the feedback path will be activated to reactivate the *GlobalOperator* module after another 10μs (using the second *GenerateDelay* module). Since the loop delay consisting of the *GlobalOperator*

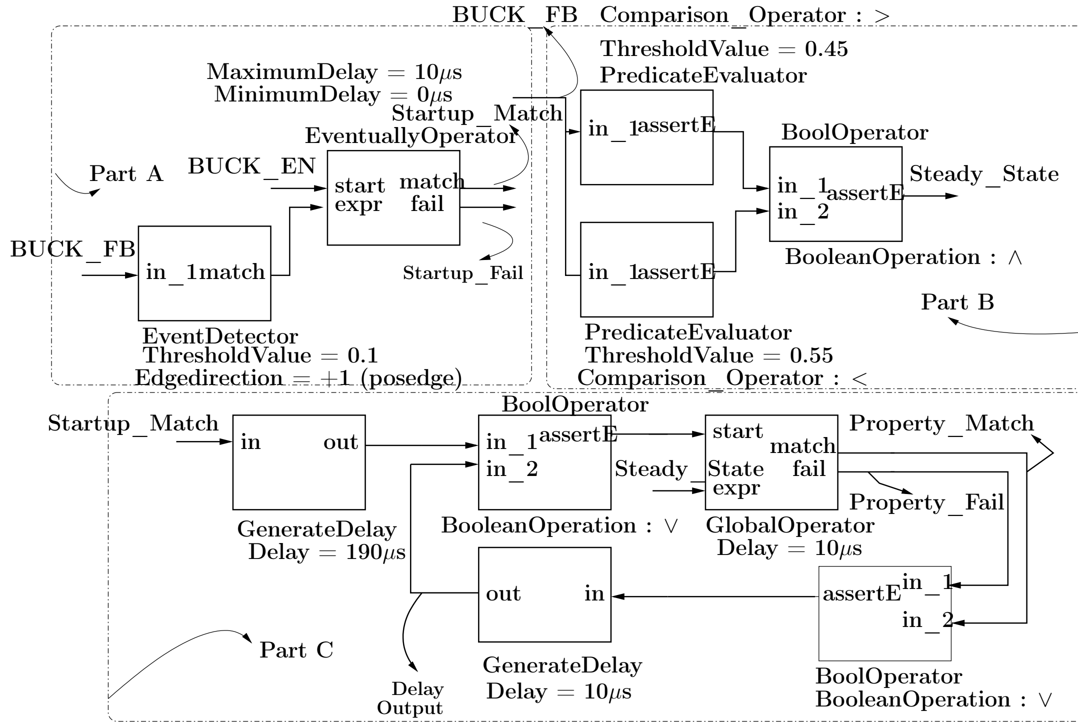


Figure 3.12: AMS-VL Realization of Example 3.4

and the second *GenerateDelay* module is upperbounded by 20μs, the constraint on sampling granularity is satisfied. ■

The following property has been checked on a PLL circuit from Freescale Inc.

Example 3.5 *When the PLL gets locked, the frequency of oscillation of pll_ refclk and pll_fdbkclk should be equal. The frequencies should remain equal till PLL remains locked.*

Figure 3.13 shows the AMS-VL realization of the Property 3.5. In Figure 3.13, *pll_ en* is the enable pin, *pll_ refclk* is the reference clock pin and *pll_fdbkclk* is the feedback clock pin of the PLL used to implement our property. Stability in the voltage of the pin *vctrl* of PLL is used to determine the locking condition. Sample code for the auxiliary function *FrequencyDetector* has been shown in Appendix A.

In Part A of Figure 3.13, the network uses and auxiliary function namely FrequencyDetector to detect the frequency of oscillation of *pll_ refclk* and *pll_fdbkclk*. In Part B of the same figure, the network detects whether the PLL has entered into the lock state by checking the voltage of the *vctrl* pin. In Part C of the figure, *ArithmeticOperator* calculates the difference of the frequency of two pins. The two *PredicateEvaluators* check whether the difference is within certain tolerance value as shown in the figure. The *indicateLock* (as detected in Part B) activates the *PredicateAssert* module and keeps it activated as long as the PLL is locked. De-

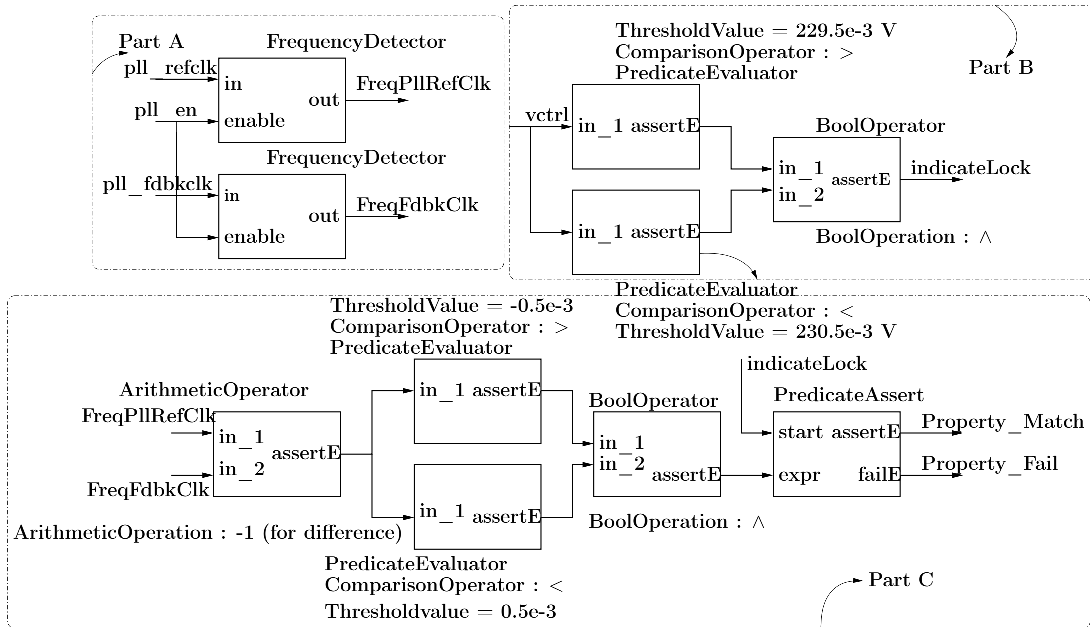


Figure 3.13: AMS-VL Realization of Example 3.5

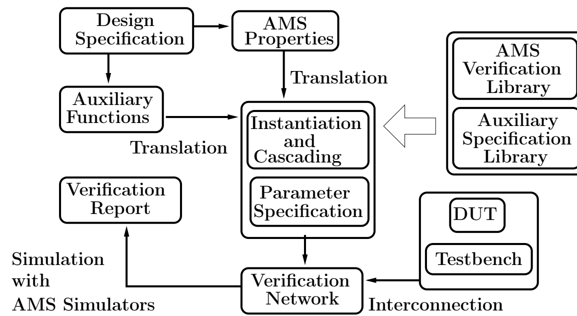


Figure 3.14: Tool Flow of AMS Verification Library

pending upon the frequency difference value, the *PredicateAssert* will accordingly either assert *assertE* or *failE*. ■

3.4. Tool Flow and Implementation Issues

Figure 3.14 (Figure 1.1 reproduced for ease of understanding) shows the tool flow for the verification of AMS behaviors using AMS-VL along with auxiliary functions. In the present version of the tool, auxiliary functions are encoded in Verilog-AMS.

The library is implemented as a package consisting of source code of the modules and the symbols in Cadence CDBA format. The library can be installed in Cadence AMS Virtuoso Environment by simply adding the library through Library Manager. Default values of parameters specified at the cell level can be overridden by specifying parameter values at the instance level which can be done

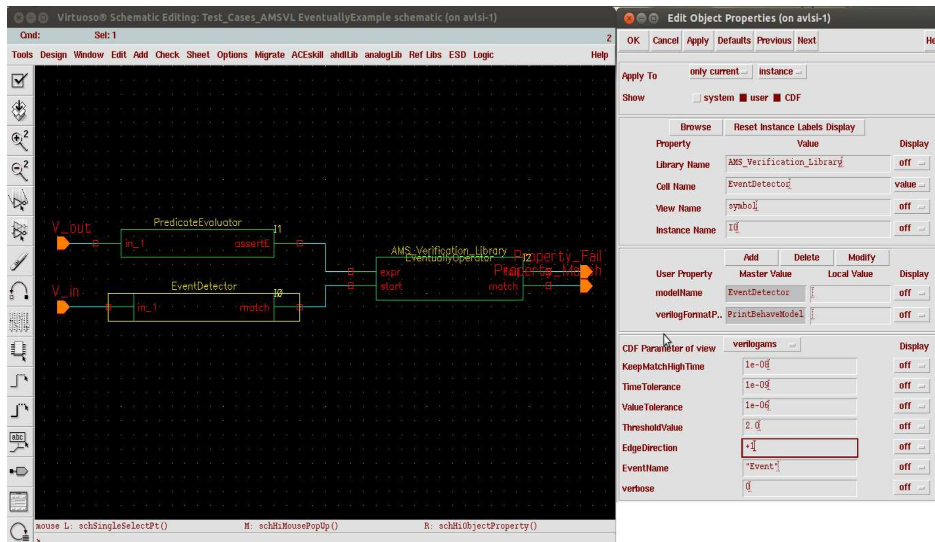


Figure 3.15: Schematic of Example 3.3

through symbols in Cadence Virtuoso. These parameters are called CDF (Component Description Format). Detailed description can be found in Cadence reference manuals [4]. We show in Figure 3.15 how symbols can be used to develop checker networks. In Figure 3.16, we show one of our test case consisting of six LDOs and associated verification networks built using AMS-VL modules.

Some of the main implementation issues are described in the following subsections.

3.4.1. Synchronization with the AMS Simulator

We use *cross_ events* to check properties. The accuracy with which the predicates are evaluated with the help of the *cross_ events* is controlled by *TimeTolearnce* and *ValueTolerance* parameters of the *cross_ events*. The *TimeTolerance* parameter specifies the maximum allowable error on the real time scale between the estimated crossing point and the true crossing point and the *ValueTolerance* parameter specifies the maximum allowable error on real value scale between estimated crossing point and the true crossing point. For example, the change in the truth value of the predicate $(V(out) > 2.0)$ can be monitored by the *PredicateEvaluator* module with the help of the following cross event. Here `match` is a logic signal which is asserted as soon as $V(out)$ crosses 2.0 volts in the positive edge direction (denoted by $+1$ in the *cross* statement below).

```
initial begin
    match = 1'b0;
 end
 always @(cross(V(out) - 2.0, +1, TimeTolerance,
```

Figure 3.16: LDO Test Case and associated Verification Networks

```
                    ValueTolerance))
    begin
        match = 1'b1;
    end
```

Consider the following property :

Example 3.6 *: After V_{in} crosses 3.0 volts, V_{out} should cross 2.5 volts sometime between 10μs and 20μs.*

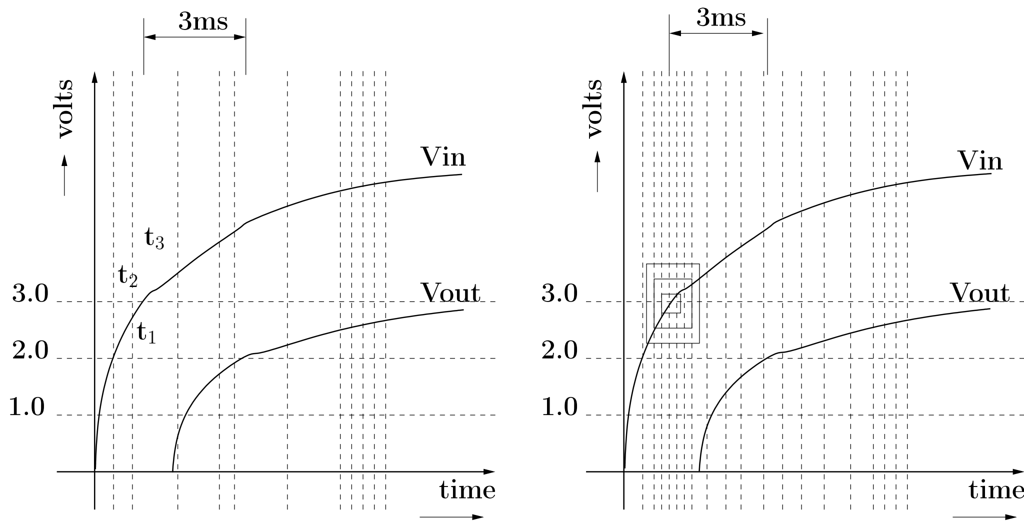


Figure 3.17: Timing Diagram of Example 3.6

In this example, monitoring the time point when V_{in} crosses 3.0V is very important. The analog simulator uses its own proprietary heuristics to place the simulation point along the timescale. As shown in the left side diagram of Figure 3.17, due to lack of sufficient simulation points near t_2 where actual crossing of 3.0V of V_{in} takes place, the event cannot be detected with sufficient accuracy which may give false report on the satisfaction / violation of the property. Point t_1 is too early and point t_3 is too away. Using `time tolerance` and `value tolerance` parameter of the *cross_ event* construct of Verilog-AMS, we can force the analog simulator to place a simulation point near the true crossing point. We show in the right side timing diagram of Figure 3.17, how analog simulator may place simulation points after using *cross_ events*.

The use of cross events creates simulation overhead because the AMS simulator has to insert additional simulation points to report the cross within the specified value and time tolerance. The time and value tolerances in the *PredicateEvaluator* and *EventDetector* modules should be carefully chosen by the user keeping in mind that over constraining may lead to degradation in simulation performance.

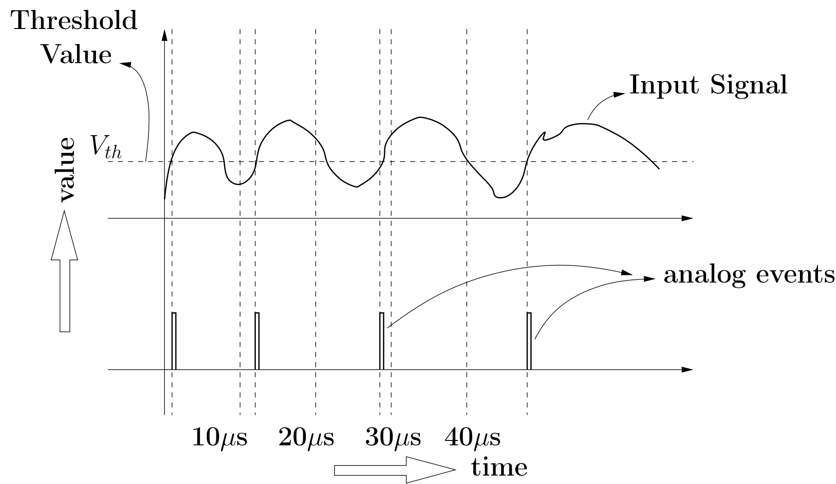


Figure 3.18: Scenario for Property Checking in Parallel Threads

3.4.2. Spawning Threads for Overlapping Matches

It is often the case that multiple matches of a property overlap in time along a simulation trace. In order to ensure that matches and violations are not missed, the AMS-VL modules need to be able to handle such overlaps. The following example illustrates one such scenario.

Example 3.7 *: After v_{in} crosses V_{th}, v_{out} should cross V'_{th} sometime between 15μs and 25μs.*

Figure 3.18 shows the v_{in} waveform, highlighting the places where it crosses V_{th}. Since successive crossings happen earlier than the time interval of the property, that is, $[15\mu s, 25\mu s]$, this is a candidate situation where overlapping matches/failures are possible. ■

In AMS-VL, overlapping matches are handled by spawning a new thread whenever a module is triggered. This is necessary only in the *Interval Operation Modules*, that is, the modules representing the temporal operators. To implement this feature, we have used the *task* construct and the *fork-join* construct of Verilog-AMS. We present a code fragment of the *EventuallyOperator* module to explain the implementation.

```
1  event trig_match;
2  event trig_fail;
3  always @(trig_match)
4          begin
5              TimeOfMatch[NumberOfStart] = $abstime;
6              match = 1'b1;
```

```
7           @(cross(($abstime - (TimeOfMatch[NumberOfStart] +
8                   KeepMatchHighTime)), +1, TimeTolerance)) ;
9           match = 1'b0;
10       end
11
12 always @(trig_fail)
13       begin
14          TimeOfFail[NumberOfStart] = $abstime;
15          fail = 1'b1;
16          @(cross(($abstime - (TimeOfFail[NumberOfStart] +
17                  KeepMatchHighTime)), +1, TimeTolerance)) ;
18          fail = 1'b0;
19       end
20
21 task my_task();
22 begin
23      fork
24         begin : test_maxdelay_time
25             #MaximumDelay;
26             -> trig_fail;
27             disable test_expr;
28         end
29         begin : test_expr
30             #MinimumDelay;
31             if (expr)
32                 ->trig_match;
33             else begin
34                 wait(expr);
35                 if(($abstime - TimeOfStart[NumberOfStart])
36                      < MaximumDelay)
37
38                     ->trig_match;
39                 else
40                     ->trig_fail;
41             end
42             disable test_maxdelay_time;
43         end
44         begin : start_again
45             @(posedge start);
46             if(NumberOfStart == MaxNumberOfStart)
47                 NumberOfStart = 0;
```

```
48                else
49                    NumberOfStart = NumberOfStart + 1;
50                TimeOfStart[NumberOfStart] = $abstime;
51                my_task;
52            end
53         join
54 end
55 endtask
56
57 always @(posedge start)
58        begin
59         if(NumberOfStart == MaxNumberOfStart)
60             NumberOfStart = 0;
61         else
62             NumberOfStart = NumberOfStart + 1;
63         TimeOfStart[NumberOfStart] = $abstime;
64         my_task;
65        end
```

Explanation of the Code Fragment :
Initially, the module waits for an occurrence of *start* (line 57 - 65) and rest part of the code remains inactive. As soon as the first *start* pulse comes, task *my_ task* is called upon. *my_ task* (line 21 - 55) uses *fork-join* construct to generate three concurrent monitoring threads. The first thread namely *test_ maxdelay_ time* (line 24 - 28) checks whether the *MaximumDelay* time has been elapsed after occurrence of *start*. The second thread namely *test_ expr* (line 29 - 43) neglects the first *MinimumDelay* time after occurrence of *start*. Then it starts looking for occurrence of *expr*. The third thread namley *start_ again*, continuously looks for another occurrence of *start* event so that no potential event is missed. This thread makes the code *reentrant*. The thread *test_ expr*, after waiting for *MinimumDelay*, if finds expr, it asserts *named event* [3] *trig_ match* else it keeps on waiting for *expr*. As soon as it gets *expr*, it checks whether *MaximumDelay* is crossed or not. If *MaximumDelay* is not crossed, it asserts *named event trig_ match* else it asserts *trig_fail* and disables the thread *test_ maxdelay_ time*. The other thread i.e. *test_ maxdelay_ time* if not disabled by *test_ expr*, and *MaximumDelay* is crossed. then asserts *trig_fail* and disables *test_ expr*. The named events are impulse of zero duration. As soon as trig_match is asserted (line 3 - 10), an impulse of duration *KeepMatchHighTime* is generated at *match* output else if *trig_fail* is asserted (line 12 - 19), then an impulse of duration *KeepMatchHighTime* is generated at *fail* output. Due to the interleaved execution of the above mentioned three threads, every potential event can be detected and potential *match/violation* of a property

can be reported. ■

3.5. Simulation Results

We studied the verification of three industrial test cases, all of which are from the power management domain. Test Case I contained six Low Drop-Out (LDO) regulators. For this circuit, monitors for 28 properties (see Appendix B.3) were developed using AMS-VL. Test Case II contained two Buck regulators, and we developed monitors for 10 properties (see Appendix B.4) using AMS-VL. Test Case III was an Integrated Circuit having four LDOs and one Buck regulator. We developed monitors for 33 properties (see Appendix B.5) in AMS-VL for this circuit. Each property required a carefully selected network of AMS-VL modules. Moreover, in all of these cases, auxiliary functions were developed and used seamlessly with the AMS-VL modules. The approximate number of *cross_events* encountered are 248 for the LDO circuit, 100 for the Buck regulator circuit and 600 for the Integrated circuit. For each of the above mentioned testcase, we assume that the testbench is given to us and the testbench has complete coverage. For detail of the testcases, please see Appendix B.

Table 3.27 shows the overhead incurred by the simulator towards handling the auxiliary functions and the AMS-VL modules. The simulations were carried out on a 2.33 GHz, Intel-Xeon server with 32GB RAM using Cadence® irun. Table 3.28 reports some details about the circuits that are used as test cases in Table 3.27.

Table 3.27: CPU Time for Simulations of Circuits

Cross Event Precision		Sim Time	CPU Time (secs)			Overhead (%)
time (sec)	value (V)	(μsec)	Design	Design + Aux Func	Design + Aux Func + Prop Check	
Test Case I: LDO Circuit						
1e-9	1e-6	700	83.89	84.51	97.12	15.77
1e-6	1e-4	700	83.89	84.51	96.01	14.45
1e-4	1e-3	700	83.89	83.51	95.42	13.75
Test Circuit II: BUCK Circuit						
1e-9	1e-6	500	90.25K	92.31K	98.05K	8.65
1e-6	1e-4	500	90.25K	92.31K	97.78K	8.35
1e-4	1e-3	500	90.25K	92.31K	97.66K	8.22
Test Case III: IC Netlist						
1e-9	1e-9	600	75.44K	75.82K	81.24K	7.69
1e-6	1e-4	600	75.44K	75.82K	81.09K	7.49
1e-4	1e-3	600	75.44K	75.82K	80.89K	7.23

The following observations may be made from the experimental results:

Table 3.28: Description of the Testcases

Test Cases	No. of Nodes	No. of Transistors	No. of Capacitors	No. of Resistors
LDO circuit	8604	2016	5166	7608
BUCK circuit	3586	4910	700	990
IC Netlist	7529	3799	3794	5567

1. The overhead of property checking is non-trivial, but the overhead becomes marginal with increase in the size of the circuits. For example, the LDOs are lightweight circuits as compared to buck regulators, and hence the overhead is more visible for LDOs as compared to buck regulators and PMUs.

2. It is interesting to see that the auxiliary functions have an insignificant contribution in the overhead. The AMS-VL modules are directly responsible for the overhead. This is largely due to the *cross events* that the AMS-VL modules introduce, thereby increasing the number of simulation points near the occurrences of those events.

3.6. Concluding Remarks

We believe that the library based verification approach will find acceptance in industrial practice. In the AMS domain, auxiliary functions appear to be significant value, and AMS-VL modules can be used seamlessly with auxiliary functions. The current version of AMS-VL is compatible with all mixed mode simulation platforms which support Verilog-AMS. Our results show that AMS-VL modules do have simulation overhead, but we believe that the online debugging capability that AMS-VL monitors provide will outweigh the simulation overhead.

Chapter 4
Verification of Simulation Relations

A hybrid system typically consists of a digital controller which interacts with an analog environment called *plant*. Embedded digital control is widely used in a variety of hybrid system domains, including automotive control, avionic control, medical instrumentation, robotics etc. One of the main challenges in designing controllers with analog environments is in modeling the interaction between the controller and the plant, which is essentially a discretized version of the control algorithm. The specification of a controller for a complex hybrid system is typically developed after analyzing the control algorithm in detail. Some of these controllers can be modeled as a *transition system (or automaton) labeled with analog predicates defined over real valued variables.* In this work, we study a symbolic method to find simulation relations between such predicate labeled transition systems.

The development of a digital controller for a hybrid system typically consists of two phases, namely:

1. *The modeling phase.* In this phase the control algorithm is typically modeled in a simulation platform (such as Simulink/stateflow) and evaluated with hybrid models of the plant. There exists formalism like hybrid automata [16] for modeling the dynamics of the plant, a significant volume of literature on formal analysis of hybrid system models [15, 16, 41, 42], and a wide arsenal of tools [62] that support such modeling styles. The modeling phase yields a high level abstract specification of the controller, which is typically an automaton that reads an abstract view of the plant and drives specific control outputs to the plant. We shall refer to this automaton as the *specification automaton.*

2. *The implementation phase.* The actual implementation of the controller starts with the *specification automaton* as the reference. For controllers that are implemented in hardware, this step is the actual circuit design step. For controllers that are implemented in a embedded computing platform, this step is a refinement step, where implementation specific details are con-

sidered. The implementation phase yields a discrete finite state machine, typically having many more states than the specification automaton. We shall refer to this state machine as the *implementation automaton*.

In order to demonstrate the difference between the specification automaton and the implementation automaton, let us consider the simple task of designing a controller for a pump that is switched on/off depending on the level of water in a tank. The plant consists of the pump and the model for the consumption pattern from the tank, that is, the plant models the dynamics for water consumption from the tank and the rate at which the water is filled up when the pump is running. Based on the analysis of the consumption patterns and the refilling rates, the control algorithm found suitable is shown in Figure 4.1(a) (same as Figure 1.2, reproduced for ease of understanding). This automaton models the following strategy:

1. P1 : *If the water level, w, is below 10 units, then the controller switches on the pump.*
2. P2 : *If the water level, w, is above 85 units, then the controller switches off the pump.*

The controller takes its decisions based on the truths of the predicates, $(w < 10)$ and $(w > 85)$. It is important to note that it does not need to see how the water level rises and falls, that is, the dynamics of consumption and refilling is not visible to the controller, though the control algorithm was designed in the modeling phase taking these dynamics into consideration. The control algorithm is designed to take its control decisions (such as switching on/off the pump) based on an abstract view of the plant that it controls. Such controllers are quite common in various application domains. We shall refer to predicates like $(w < 10)$ as *predicates over real variables* (PORVs). The formal definition of PORVs has been presented later.

The implementation of the controller needs to take into consideration various other aspects that may require refinement of the specification. For example, in order to accommodate discrepancies in the sensing of the water level, an implementation may choose to switch on the pump whenever the water level falls below 15 units. If the role of the controller is to ensure that the water level remains above 10 units at all times, then this is an acceptable choice. On the other hand, if a controller does not switch on the pump even when the water level is below 10 units, then the controller is not acceptable. This definition of acceptability is based on our intuitive understanding that the role of the controller is to maintain the water level above 10 units, though this does not formally follow from the specification automaton. We shall return to this discussion later.

Figure 4.1(b) and Figure 4.1(c) show two implementation strategies for the water tank controller. These controllers have additional states corresponding to other aspects of control as explained below:

1. The *Off* state of the pump has two variants, namely *Hibernate* and *Deep Sleep*. Once the pump enters the *Hibernate* state (shown as (*Off,1*)), where the pump is *Off*, it starts a timer. If the water level goes below the threshold before the timer completes its count, then the pump is switched on. But if the water level does not go below the threshold before and timer completes its count, then the controller goes to the *Deep Sleep* state (shown as (*Off,2*)), where the pump is still *Off*. From the *Deep Sleep* state, the pump will go to *On* state once the water level falls below the threshold.

2. The *On* state of the pump too has two variants. When the pump is switched on, the rate of water filling may be kept high for a certain time which will be indicated by a timer signal. Let us name this state *High Rate* (shown as (*On,1*)) where the pump is on. If the upper threshold of the water level is reached before the timer completes its count, the controller switches off the pump. If the timer completes its count before the upper threshold is reached, then the filling rate is lowered and the controller moves to the *Low Rate* state (shown as (*On,2*)). Once the water level reaches the upper threshold, the pump is switched *Off*.

It may be noted that the controllers shown in the figures are not actually concerned with timing. The timer are external – the controller only reads the timeout signals and asserts the signals for setting the timers.

Our task is to determine whether the implementation automata shown in Figure 4.1(b) and Figure 4.1(c) are acceptable with respect to the specification automaton. It is important to note that the PORVs labeling the specification automaton are not identical to the PORVs labeling the implementation automata. Therefore, if we consider sequential equivalence between these automata with the propositions and PORVs as labels, then none of the two implementation automata are equivalent to the specification automaton.

Our main point of difference with sequential equivalence checking of labeled transition systems is in studying the relation between the PORVs. For example any valuation of w that satisfies the PORV $(w < 10)$ also satisfies the PORV $(w < 15)$, and no valuation of w that refutes the PORV $(w < 15)$ satisfies the PORV $(w < 10)$. Therefore if we maintain the water level above 15 units, it automatically guarantees that the water level stays above 10 units. Such an intuitive understanding leads us to conclude that the implementation automata in

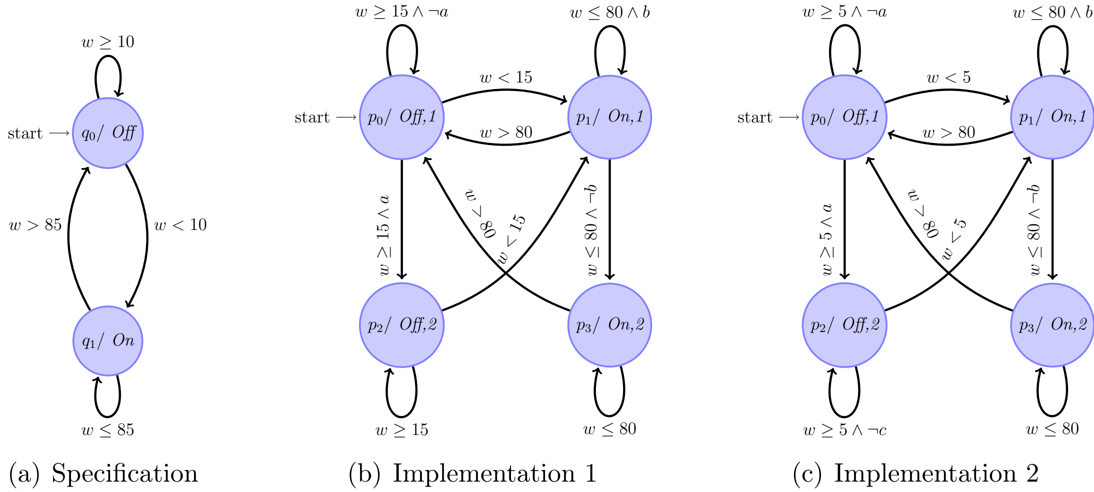


Figure 4.1: A Specification and Two Implementations

Figure 4.1(b) is acceptable with respect to the specification automaton, but the implementation automata in Figure 4.1(c) is not acceptable.

Such intuitive definition of acceptability is not necessarily correct. For example, it could be the case that for some special type of pump it is mandatory to keep the pump off for some specific period of time before it can be switched on again. The consumption pattern for the tank may be such that the time taken for the water level to fall from 85 to 10 is less than the this required shutoff period. We may therefore choose to relax the guard $(w < 10)$ to $(w < 5)$ in order to allow more time for the pump to come out of its shutoff mode. This is a requirement that is not automatically captured from the specification automaton, just as our intuitive understanding that the water level has to be maintained between 10 and 85 units does not follow from the specification automaton. Therefore the permissible direction in which the PORVs in the specification can be refined in the implementation has to be explicitly given by the user, for us to be able to determine computationally whether an implementation automaton is acceptable with respect to the given specification automaton. We shall refer to these as *refinement directives*.

This chapter defines the formal models for the specification automaton with refinement directives and the implementation automaton. The chapter presents the definition of simulation relations between the specification and implementation automata, and presents formal methods for computing such relations.

The chapter is organized as follows: In Section 4.1, we describe the formal model of computation and formally define *path based simulation relation* in the context of PORV labeled transition systems. In Section 4.2 we explain the transformation steps required to map the proposed simulation relation finding problem

to the Kanellakis-Smolka algorithm. Section 4.3 concludes the chapter.

4.1. Simulation Relation Finding Methodology

Here we present our model of computation formally. Then, we explain the proposed algorithm described for simulation relation finding.

4.1.1. Formal Model of Computation

In this section we present the formal definition of simulation relations for predicate labeled transition systems.

Definition 4.1.1 [Predicate Over Real Variables] *: A predicate over a set of real valued variables* $\mathbf{V} = \{x_1, x_2, \ldots, x_n\}$ *is defined by a n tuple* $\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle \in \mathbb{R}^n$ *and a relational operator* $\sim \in \{>, \geq\}$*. The PORVs represent inequalities of the form* $(\alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n + c) \sim 0$ *where c is any constant and* $c \in \mathbb{R}$. ■

Definition 4.1.2 [LTS with PORV label] *: A* controller *is a labeled transition system (LTS) defined as a 7-tuple :*

$$\mathcal{G} = \langle\, Q,\, I,\, \delta,\, Q_0,\, \mathcal{AP},\, var,\, \mathcal{L}\, \rangle$$

where:

- Q *is the set of the states of the controller.*
- $Q_0 \subseteq Q$ *is the set of initial states.*
- $\mathcal{AP}$ *is the set of atomic propositions (labels).*
- var *is the set of real variables.*
- $I = I_B \cup I_{PORV}$ *is the set of inputs of the controller, where* I_B *is the set of Boolean signals and* I_{PORV} *is the set of PORVs over* var*.*
- $\delta \subseteq Q \times 2^I \times Q$ *is the transition relation.*
- $\mathcal{L} : Q \rightarrow 2^{\mathcal{AP}}$ *is a function for labeling the states in* Q *with propositions in* $\mathcal{AP}$*.* ■

In our problem, both the specification and implementation are given as LTSs with PORV labels. The implementation is a *refinement* of the specification. In digital designs, refinement means that every run of the implementation is also a run of the specification. In our case, the PORVs labeling the specification and the PORVs labeling the implementation are not necessarily the same and hence

we cannot compare two runs by studying their labels. The refinement directives, as defined below, give us the necessary condition for comparing the runs of the specification and the implementation.

The LTS for the *specification* is annotated with *refinement directives* that indicate the admissible directions in which input PORVs can be *strengthened* or *relaxed* in the *implementation*. For example, in Figure 4.1(a), the PORV labeling the transition from *Off* to *On* state i.e $w < 10$ can be *weakened* to $w < 15$ in an implementation (to allow aberration for reading the value of w) and the PORV $w \geq 10$ labeling the self loop at the *Off* state has to be *strengthened* to $w \geq 15$ as shown in Figure 4.1(b). This does not follow automatically from the automaton of Figure 4.1(a) and needs to be explicitly specified by the user. Formally, a PORV α is stronger than a PORV β if $\alpha \Rightarrow \beta$ and similarly if PORV α is weaker than PORV β, then $\beta \Rightarrow \alpha$. Formally, a *refinement directives*, γ is a function of the form :

$$\gamma : \mathcal{H} \rightarrow \{W, S\}$$

where $\mathcal{H}$ is the guard condition for state transition s_k to s_j i.e. $s_k \xrightarrow{\mathcal{H}} s_j$, $\mathcal{H} = \bigwedge \alpha_l$, where $\alpha_l \in I^s$. Here W presents *weakening* and S presents *strengthening*. We use $\mathcal{G}^s = \langle\ Q^s,\ I^s,\ \delta^s,\ Q^s_0,\ \mathcal{AP}^s,\ var^s,\ \mathcal{L}^s,\ \gamma\ \rangle$ and $\mathcal{G}^i = \langle\ Q^i,\ I^i,\ \delta^i,\ Q^i_0,\ \mathcal{AP}^i,\ var^i,\ \mathcal{L}^i\ \rangle$ for the LTS of the *specification* and the LTS of the *implementation* respectively.

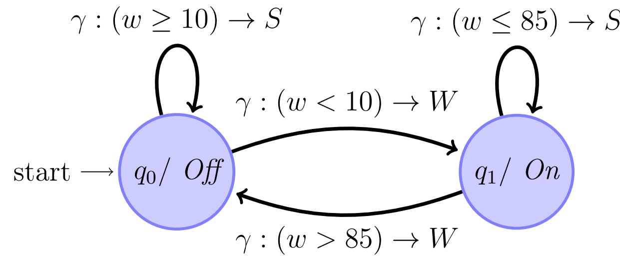
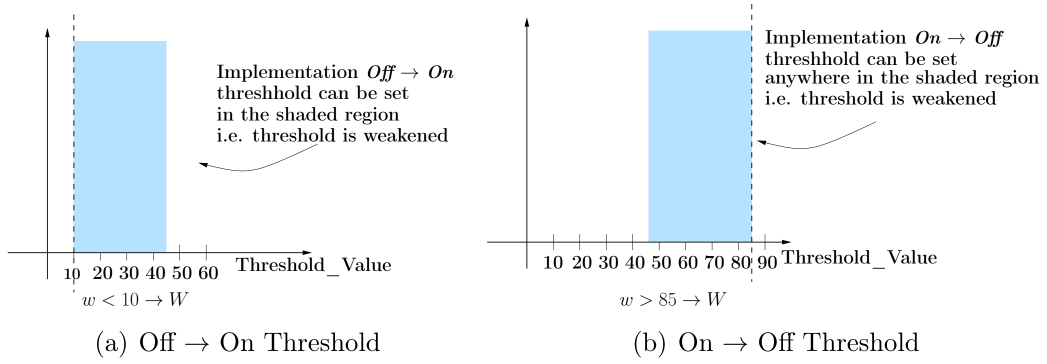


Figure 4.2: Specification LTS Annotated with Refinement Directives γ

Example 4.1 *: As an example, we define the constituents of the tuple corresponding to the specification LTS in Figure 4.2 as follows:*

- $Q^s = \{q_0,\ q_1\}$
- $Q^s_0 = \{q_0\}$
- $\mathcal{AP}^s = \{\text{Off, On}\}$
- $var^s = \{w\}$



(a) Off → On Threshold (b) On → Off Threshold

Figure 4.3: Zone of Interest of Implementation PORVs

- $I_B^s = \emptyset$ *and* $I_{PORV}^s = \{w < 10,\ w > 85,\ w \leq 85,\ w \geq 10\}$
- δ^s *is as illustrated in the Figure 4.1(a).*
- $\mathcal{L}^s(q_0) = \{\text{Off}\}$ *and* $\mathcal{L}^s(q_1) = \{\text{On}\}$
- γ *:* $\{(w < 10) \rightarrow W\}$, $\{(w > 85) \rightarrow W\}$, $\{(w \leq 85) \rightarrow S\}$ *and* $\{(w \geq 10) \rightarrow S\}$. *The weakening zone of the PORVs* $w < 10$ *and* $w > 85$ *are shown in the Figure 4.3(a) and Figure 4.3(b) respectively.* ■

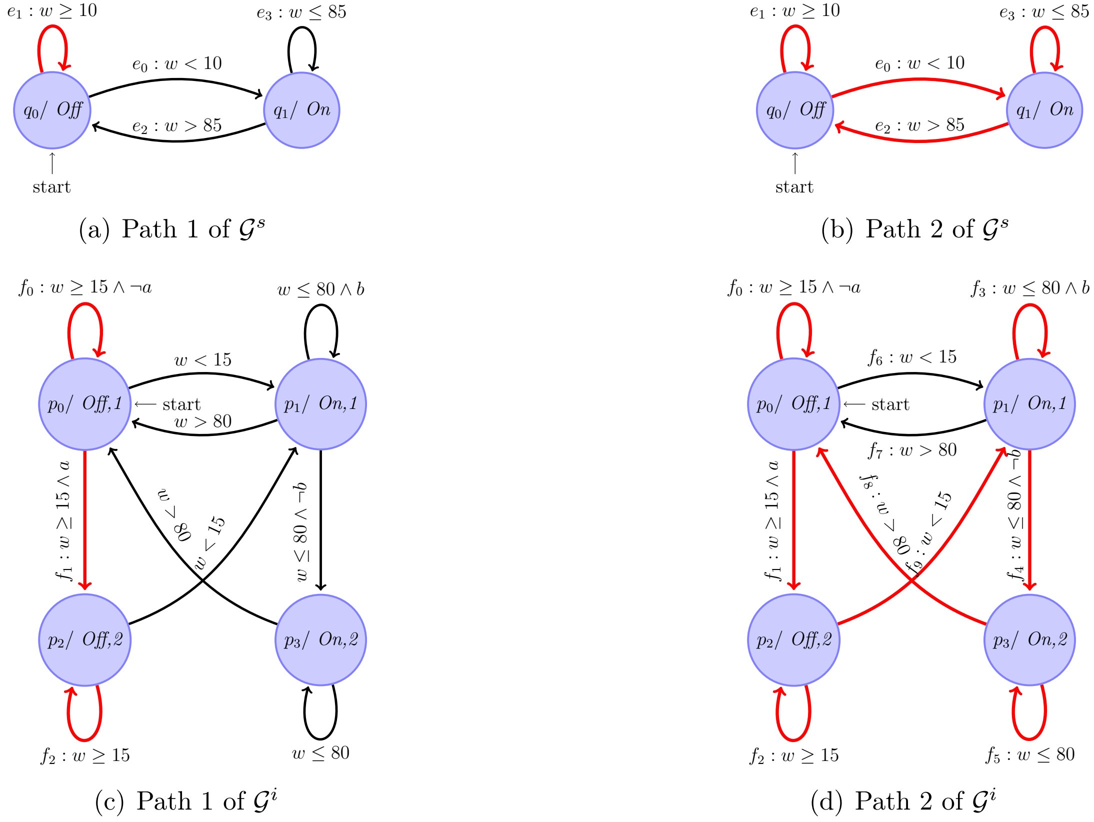
Definition 4.1.3 [Path Fragment in LTS] *A finite path fragment* π' *of a LTS is a finite state sequence* $q_0, e_0, q_1, e_1, \ldots, q_n$ *such that* $q_i \xrightarrow{e_k} q_{i+1}$ $\forall 0 \leq i \leq n$*, where* $n \geq 0$ *and* $k \geq 0$*,* e_k *is the transition guard condition. An infinite path fragment* π *is an infinite state sequence* $q_0, e_0, q_1, e_1 \ldots\ldots$ *such that* $q_i \xrightarrow{e_k} q_{i+1}$ $\forall i > 0$ *and* $k \geq 0$. ■

Definition 4.1.4 [Maximal and Initial Path Fragment in LTS] *A maximal path fragment is either a finite path fragment that ends in a terminal state or an infinite path fragment. If a path fragment starts in an initial state i.e.* $q_0 \in Q_0$*, then it is called initial path fragment.*

A *maximal path fragment* is a path fragment that cannot be prolonged : either it is infinite or it is finite but ends in a state from where no more transition is possible. ■

Definition 4.1.5 [Path in LTS] *: A path of a LTS* $\mathcal{G}$ *is an initial, maximal path fragment.*

We indicate a path in *specification* LTS by π^s and a path in *implementation* LTS by π^i. We have shown paths of *implementation* LTS in Figure 4.4(c), and Figure 4.4(d). We show corresponding paths of *specification* LTS in Figure 4.4(a),



(a) Path 1 of $\mathcal{G}^s$

(b) Path 2 of $\mathcal{G}^s$

(c) Path 1 of $\mathcal{G}^i$

(d) Path 2 of $\mathcal{G}^i$

$\pi_1^i = \{p_0, f_0, p_0, f_1, p_2, f_2, p_2, \dots\}$ in 4.4(c) and corresponding $\pi_1^s = \{q_0, e_1 q_0, \dots\}$ in 4.4(a). .

Figure 4.4: Paths in Specification and Implementation LTS

and in Figure 4.4(b) respectively. Let *Paths*($\mathcal{G}^s$) denote the set of all paths in $\mathcal{G}^s$ and *Paths*($\mathcal{G}^i$) denote the set of all paths in $\mathcal{G}^i$.

Let $\Sigma = I_B \cup var \cup \mathcal{AP}$ denote the set of the variables of the controller. $I_B \cup var$ contains the input variables and $\mathcal{AP}$ represents the set of the outputs asserted by the controller. We use Σ^s and Σ^i to denote the set of variables for *specification LTS* and *implementation LTS* respectively.

Definition 4.1.6 [Signal Trace] *: A* signal trace *or simply a* trace *of the infinite path fragment* $\pi = q_0, e_0, q_1, e_1, \dots$ *of a LTS,* σ*, is an infinite sequence* $\sigma_0, \sigma_1, \dots$ *where each* $\sigma_i \in 2^{I_B} \times \mathbb{R}^{|var|} \times 2^{\mathcal{AP}}$*. In other words,* σ *is an infinite sequence of valuations of the variables in* Σ *or the induced finite or infinite word over the alphabet* $2^{I_B} \times \mathbb{R}^{|var|} \times 2^{\mathcal{AP}}$*.* ∎

We use σ^s and σ^i to denote a trace of a *specification LTS* and a trace of an *implementation LTS* respectively. We denote the elements of an infinite sequence

in the *specification LTS* by σ_i^s and of an *implementation LTS* by σ_k^i. Since the valuations of the variables in *var* are real valued, there can be uncountably infinite traces in the specification LTS as well as in the implementation LTS.

Example 4.2 *: Following is an example trace of the* implementation *LTS of Figure 4.4(d) and the path generated is* $p_0, f_0, p_0, f_1, p_2, f_9, p_1, f_3, p_1, f_3, p_1, f_4, p_3, f_5, p_3, f_8, p_0, \ldots$ *:*

$$\sigma^i = \underbrace{\{w = 20, Off\}}_{\sigma_0^i},\ \underbrace{\{w = 16, Off\}}_{\sigma_1^i},\ \underbrace{\{w = 15.5, Off\}}_{\sigma_2^i},\ \underbrace{\{w = 14, On\}}_{\sigma_3^i},$$
$$\underbrace{\{w = 9, On\}}_{\sigma_4^i},\ \underbrace{\{w = 60, On\}}_{\sigma_5^i},\ \underbrace{\{w = 70, On\}}_{\sigma_6^i},\ \underbrace{\{w = 78, On\}}_{\sigma_7^i},\ \underbrace{\{w = 81, Off\}}_{\sigma_8^i}, \ldots$$

Definition 4.1.7 [Sim(π^i,σ)] *: A trace* $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots$ *models a path* $\pi^i = q_0^i, e_0^i, q_1^i, e_1^i, \ldots$ *of* $\mathcal{G}^i$ *if the following conditions hold good :*

1. *for all k, the set of atomic propositions that are true in* σ_k *exactly match with the set of atomic propositions that are true in state* q_k^i.
2. *for all k, the valuation of real valued variables in each* σ_k*, make the transition guard condition* α *of* e_k^i *true where* $q_k^i \xrightarrow{\alpha} q_{k+1}^i$.

In Example 4.2, we show a trace σ which generates a path π^i in the *implementation* $\mathcal{G}^i$ of Figure 4.4(d) such that *Sim(*π^i*,*σ*)* is true. We denote the set of all traces such that *Sim(*π^i*,*σ*)* is true by *Traces(*$\mathcal{G}^i$*)*.

Definition 4.1.8 [Sim(π^s,σ, γ)] *: A trace* $\sigma = \sigma_0, \sigma_1, \sigma_2, \ldots$ *models a path* $\pi^s = q_0^s, e_0^s, q_1^s, e_1^s, \ldots$ *of* $\mathcal{G}^s$ *under a given refinement directives* γ *if the following conditions hold good :*

1. *for all k, the set of atomic propositions that are true in* σ_k *exactly match with the set of atomic propositions that are true in state* q_k^s.
2. *for all k, the valuation of real valued variables in each* σ_k *is such that they satisfy the weakening or strengthening refinement directive on the transition guard condition* β *of* e_k^s *where* $q_k^s \xrightarrow{\beta} q_{k+1}^s$.

The Condition 2 of Definition 4.1.8 implies the following :

1. If the *refinement directives* is *W*, then the valuation of the real valued variables will not make the guard conditions of *specification* LTS *true* in proper sense, rather, the *refinement directives* γ will allow some additional room for the trace to satisfy the guard condition to trigger the associated transition.

2. If the *refinement directives* is S, then the valuation of the real valued variables will not make the guard conditions of *specification* LTS *false* in proper sense.

Consider the trace σ^i of Example 4.2. For the trace component σ^i_3, the valuation of real valued variable w is 14 which does not make the labeling predicate e_0 of Figure 4.4(b) true in proper sense. But if we bring the *refinement directives* in the scenario, then as shown in Figure 4.2, e_0 can be *weakened* and hence, the value of w satisfy e_0 under the specified γ. Hence, it enables the transition from q_0(*Off State*) to q_1(*On State*). Similar is the case for the trace component σ^i_8 which under specified γ (as shown in Figure 4.2) satisfies the labeling predicate e_2 and hence enables the transition from q_1(*On State*) to q_0(*Off State*).

Definition 4.1.9 **[$\mathcal{G}^i \preceq \mathcal{G}^s$]** *: $\mathcal{G}^i \preceq \mathcal{G}^s$ with respect to a given refinement directives γ iff forall traces $\sigma \in$ Traces($\mathcal{G}^i$), Sim(π^i,σ) true implies there exists a path $\pi^s \in$ Paths($\mathcal{G}^s$) such that both the conditions of the Definition 4.1.8 holds good i.e. Sim(π^s,σ, γ) is true. Precisely, $\forall\sigma \in$ Traces($\mathcal{G}^i$), Sim(π^i,σ) $\Rightarrow \exists\pi^s \in$ Paths($\mathcal{G}^s$), Sim(π^s,σ, γ) is true.*

Since the specification and implementation have uncountably infinite traces, it is nontrivial to verify whether $\mathcal{G}^i \preceq \mathcal{G}^s$. Algorithms like Kanellakis-Smolka and Paige-Tarjan rely on a fix-point convergence which is guaranteed by the finite nature of the state space. The main contribution of this chapter is to define a notion of path-based simulation equivalence between our PORV labeled transition systems and prove that this relation is necessary and sufficient for the relation $\mathcal{G}^i \preceq \mathcal{G}^s$.

Definition 4.1.10 **[Sim^A(π^i,π^s,γ)]** *: A path $\pi^s \in$ Paths($\mathcal{G}^s$) simulates a path $\pi^i \in$ Paths($\mathcal{G}^i$) if the following conditions hold good :*

1. *for all k, the set of atomic propositions that are true in q^i_k and the set of atomic propositions that are true in q^s_k exactly match i.e. $\mathcal{L}^i(q^i_k) = \mathcal{L}^s(q^s_k)$.*
2. *for all k, $q^i_k \xrightarrow{\alpha} q^i_{k+1}$ and $q^s_k \xrightarrow{\beta} q^s_{k+1}$, the refinement directives γ holds good between the guard conditions α and β.*

For example, the path π^s of Figure 4.4(b) and path π^i of Figure 4.4(d) is related by the *Sim^A(π^i,π^s,γ)*.

Definition 4.1.11 **[$\mathcal{G}^i \overset{A}{\preceq} \mathcal{G}^s$]** *: $\mathcal{G}^i \overset{A}{\preceq} \mathcal{G}^s$ iff $\forall\pi^i \in$ Paths($\mathcal{G}^i$), $\exists\pi^s \in$ Paths($\mathcal{G}^s$) such that Sim^A(π^i,π^s,γ) holds good.*

Theorem 4.1.1 **[Simulation Relation]** *: $\mathcal{G}^i \preceq \mathcal{G}^s$ iff $\mathcal{G}^i \overset{A}{\preceq} \mathcal{G}^s$.*

Proof :
Suppose, $\mathcal{G}^i \overset{A}{\preceq} \mathcal{G}^s$. By Definition 4.1.11, each path $\pi^i_k \in$ Paths($\mathcal{G}^i$) has a corresponding simulating path $\pi^s_j \in$ Paths($\mathcal{G}^s$) such that Sim^A(π^i,π^s,γ) holds good. Further by Definition 4.1.10, for each such path π^i_k and π^s_j, the set of atomic propositions labeling the states of the paths match and the refinement directives, γ, holds good between the guard conditions. Hence, a trace $\sigma_l \in$ Traces($\mathcal{G}^i$) such that Sim(π^i_k, σ_l), will also make the same set of atomic propositions true in π^s_j as π^i_k and as γ holds, the values of real valued variables in σ_l will satisfy the guard conditions of π^s_j. Arguing similarly, for every such $\sigma_l \in$ Traces($\mathcal{G}^i$) such that Sim(π^i_k,σ_l) will make Sim(π^s_j,σ_l, γ) true i.e. in other words, $\forall\sigma_l \in$ Traces($\mathcal{G}^i$), Sim(π^i_k,σ_l) $\Rightarrow \exists\pi^s_j \in$ Paths($\mathcal{G}^s$) such that Sim(π^s_j,σ_l, γ) is true for given γ, which is Definition 4.1.9 and hence $\mathcal{G}^i \preceq \mathcal{G}^s$.

For the other direction, suppose $\mathcal{G}^i \preceq \mathcal{G}^s$. Consider a trace $\sigma_l \in$ Traces($\mathcal{G}^i$) such that Sim(π^i_k,σ_l) for any k. By Definition 4.1.9, Sim(π^s_j,σ_l,γ) should hold good. This immediately implies (a) $\mathcal{L}^i(q^i_m) = \mathcal{L}^s(q^s_m)$ and (b) the refinement directives γ between the transition guard conditions hold good (as Sim(π^s_j,σ_l,γ) and Sim(π^i_k,σ_l) both are true), i.e. *Sim^A(π^i_k,π^s_j,γ)* is true. Arguing similarly, this holds for any $\sigma_l \in$ Traces($\mathcal{G}^i$) such that Sim(π^i_k,σ_l) and Sim(π^s_k,σ_l,γ) are true. This implies that for every path $\pi^i_k \in$ Paths($\mathcal{G}^i$), there exists a path $\pi^s_j \in$ Paths($\mathcal{G}^s$) such that Sim^A(π^i_k,π^s_j,γ) is true. This is the Definition 4.1.11 and hence $\mathcal{G}^i \overset{A}{\preceq} \mathcal{G}^s$. ■

The above theorem proves that checking $\mathcal{G}^i \overset{A}{\preceq} \mathcal{G}^s$ is sufficient to find out the *simulation relation* and we extend the KS algorithm to check that. We define two Boolean operators namely *Smooth* and *Rename* [22] which will be used later in the implementation of the proposed *symbolic simulation relation* finding algorithm.

Definition 4.1.12 [Smooth Function] *: Let f be a Boolean function. The smoothing of f by $X = (x_{i_1}, x_{i_2}, \ldots, x_{i_p})$ is defined as*

$$\begin{aligned}
Smooth(x_{i_1}, x_{i_2}, \ldots, x_{i_p})(f) &= Smooth_{x_{i_1}} \circ \cdots \circ Smooth_{x_{i_p}}(f)\\
Smooth_{x_{i_j}}(f) &= f_{x_{i_p}} + f_{\overline{x_{i_p}}}\\
f_{x_i}(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_r) &= f_{x_i}(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_r)\\
f_{\overline{x_i}}(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_r) &= f_{x_i}(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_r)
\end{aligned}$$

Logically, the *smoothing* operator performs existential quantification on the smoothed variables : $Smooth_{(x_{i_1}, x_{i_2}, \ldots, x_{i_p})}(f) = \exists x_{i_p} \ldots \exists x_{i_p}(f)$

Definition 4.1.13 [Rename Function] *: Let f be a Boolean function. Rename of array $X = [x_1, x_2, \ldots, x_p]$ by array $Y = [y_1, y_2, \ldots, y_p]$ in f noted as $Rename_{Y \leftarrow X}$ is defined as*

$$[Y \leftarrow X]f \equiv_{def} Rename_{Y \leftarrow X}\Big(\bigwedge_{i \in p}(y_i \Longleftrightarrow x_i) \wedge f\Big)$$

Given two LTS $\mathcal{G}^s$ and $\mathcal{G}^i$, our problem is to check whether $\mathcal{G}^i \preceq \mathcal{G}^s$. The following points highlight the difference between this problem and the related problems discussed in Section 2.3.

- Our *simulation relation* finding problem is not concerned with the continuous evolution of the real valued variables over time. This observation distinguishes our problem from the body of literature on equivalence checking of hybrid systems.

- In our *simulation relation* finding problem, the PORVs used in the specification LTS are not the same as the PORVs used in the implementation LTS. However they are related by virtue of being defined over the same set of real valued variables. Therefore, the *simulation relation* task must consider the relation between the PORVs of specification LTS and the implementation LTS which separates our problem from standard *simulation relation* finding problem of two state machines defined over atomic propositions.

To the best of our knowledge, the simulation relation finding problem presented in this work has not been studied in existing literature.

4.2. Methodology and Tool Flow to find Simulation Relation

Let us be given the specification LTS $\mathcal{G}^s$ and the implementation LTS $\mathcal{G}^i$ respectively. δ^s be the transition relation of $\mathcal{G}^s$ and δ^i be the transition relation for $\mathcal{G}^i$. Let $p^i_{\mathcal{G}^s} \in \mathcal{AP}^s$ be the atomic propositions that label *specification LTS* states and $p^k_{\mathcal{G}^i} \in \mathcal{AP}^i$ be the atomic propositions that label *implementation LTS* states. Let Q^s_0 be the set of initial states of $\mathcal{G}^s$ and Q^i_0 be the set of initial states of $\mathcal{G}^i$ respectively.

4.2.1. Pre-Process Steps

Let $ValidStatePairsAtoms$ be the set of states which contains those pairs of states from $\mathcal{G}^i$ and $\mathcal{G}^s$ whose labeling atomic propositions matches. The step has been depicted graphically in the Figure 4.5. Formally,

$$\begin{aligned} ValidStatePairsAtoms = &\quad \bigwedge_k \Big((p^k_{\mathcal{G}^i} \times p^k_{\mathcal{G}^s}) \vee (\overline{p^k_{\mathcal{G}^i}} \times \overline{p^k_{\mathcal{G}^s}}) \Big) \\ = &\quad \bigwedge_k \big(p^k_{\mathcal{G}^i} \Leftrightarrow p^k_{\mathcal{G}^s} \big) \\ = &\quad \bigwedge_k \big(p^k_{\mathcal{G}^i} \odot p^k_{\mathcal{G}^s} \big) \end{aligned}$$

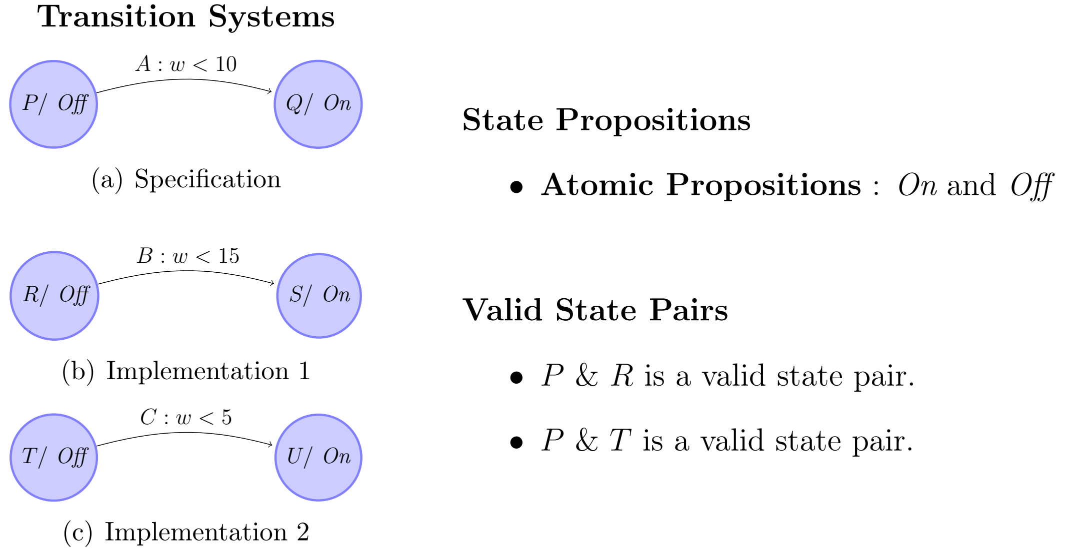


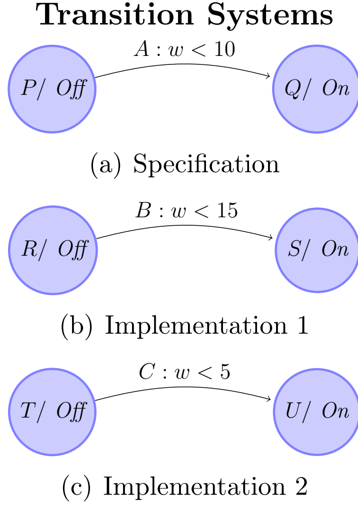
Figure 4.5: Pre-Process Step 1

We explain the idea with the help of the example shown in Figure 4.5. There are two states in the *specification* and two states in each of the two *implementations*, one state is labeled with *On* and another state is labeled with *Off*. Since, state pairs $\{P, R\}$ and $\{P, T\}$ are labeled with same atomic propositions namely *Off*. hence they are *Valid State Pairs* w.r.t atomic propositions. Arguing in a similar way, state pairs $\{Q, S\}$ and $\{Q, U\}$ are labeled with atomic proposition namely *On* and hence they form another set of *Valid State Pairs.*

A state $q_i^s \in \mathcal{G}^s$ and a state $q_k^i \in \mathcal{G}^i$ may not be consistent with each other even if they match in the atomic proposition labels because of the violation of the weakening or strengthening (refinement directives γ) of guard condition of the outgoing edges. The *simulation relation* finding algorithm sees the PORVs as just propositions and hence incapable of detecting such states where such violation takes place. The process has been depicted in the Figure 4.6. In order to eliminate such states where the refinement directives γ for the guards of the edges are violated, we do the following :

- We take $I^s_{PORV} \cup I^i_{PORV}$ i.e union of all the PORVs labeling $\mathcal{G}^s$ and $\mathcal{G}^i$ and use an SMT solver like MALL [68] to compute the set of minimal unsatisfiable cores. Let us name the minimal unsatisfiable cores as *UnSATCore*.

- Let $P^i_{\mathcal{G}_s} \in I^s_{PORV}$ which labels the guard α of an edge of $\mathcal{G}^s$ and $P^k_{\mathcal{G}_i} \in I^i_{PORV}$ which labels the guard β of an edge of $\mathcal{G}^i$. Now depending upon the annotation of α in the refinement directives γ i.e whether they are W or S annotated, $(q_k^i, q_i^s) \in ValidStatePairsAtoms$ will be indeed a valid state pair w.r.t labeling PORVs, if the following conditions hold good :



Minimal Unsat Cores

- **Involving** A **and** B : $A \wedge \neg B$
- **Involving** A **and** C : $\neg A \wedge C$

Valid State Pairs

- P & R is a valid state pair.
- P & T is not a valid state pair.

Figure 4.6: Pre-Process Step 2

– If α is S annotated, then

$$\bigwedge_k P^k_{\mathcal{G}_i} \Rightarrow \bigwedge_j P^j_{\mathcal{G}_s} \text{ is TRUE}$$

$$\equiv \left(\neg\left(\bigwedge_k P^k_{\mathcal{G}_i}\right) \vee \left(\bigwedge_j P^j_{\mathcal{G}_s}\right)\right) \text{ is TRUE}$$

hence,

$$\neg\left(\neg\left(\bigwedge_k P^k_{\mathcal{G}_i}\right) \vee \left(\bigwedge_j P^j_{\mathcal{G}_s}\right)\right) \text{ is Unsatisfiable}$$

$$\equiv \left(\bigwedge_k P^k_{\mathcal{G}_i} \wedge \neg\left(\bigwedge_j P^j_{\mathcal{G}_s}\right)\right) \text{ is Unsatisfiable}$$

$$\equiv \bigvee_j \left(\bigwedge_k P^k_{\mathcal{G}_i} \wedge (\neg P^j_{\mathcal{G}_s})\right) = stateImplication^S \text{ is Unsatisfiable}$$

For the state pair to be valid, the $stateImplication^S \in UnSATCore$ and $(q^i_k, q^s_i) \in ValidStatePairs$.
But if $stateImplication^S \notin UnSATCore$, then we discard that state pair.

– If α is W annotated, then

$$\bigwedge_k P^k_{\mathcal{G}_s} \Rightarrow \bigwedge_j P^j_{\mathcal{G}_i} \text{ is TRUE}$$

$$\equiv \left(\neg\left(\bigwedge_k P^k_{\mathcal{G}_s}\right) \vee \left(\bigwedge_j P^j_{\mathcal{G}_i}\right)\right) \text{ is TRUE}$$

hence,

$$\neg\left(\neg\left(\bigwedge_k P^k_{\mathcal{G}_s}\right) \vee \left(\bigwedge_j P^j_{\mathcal{G}_i}\right)\right) \text{ is Unsatisfiable}$$

$$\equiv \left(\bigwedge_k P^k_{\mathcal{G}_s} \wedge \neg\left(\bigwedge_j P^j_{\mathcal{G}_i}\right)\right) \text{ is Unsatisfiable}$$

$$\equiv \bigvee_j \left(\bigwedge_k P^k_{\mathcal{G}_s} \wedge (\neg P^j_{\mathcal{G}_i})\right) = stateImplication^W \quad \text{is Unsatisfiable}$$

For the state pair to be valid, the $stateImplication^W \in UnSATCore$ and $(q^i_k, q^s_i) \in ValidStatePairs$.
But if $stateImplication^W \notin UnSATCore$, then we discard that state pair.

We explain the idea with the help of the example shown in Figure 4.6. We take union of the input predicates for *Specification* and *Implementation 1* i.e $\{w < 10\} \cup \{w < 15\}$ and supply it to MALL. We get the *UnSATCore* $A \wedge \neg B$. Now as explained in the Example 4.1, PORV A is a W labeled i.e in the Implementation 1, the *threshold value* for this PORV can be increased (i.e. weakened). This implies that Specification input PORV is stronger and its *TRUTH* should imply the *TRUTH* of Implementation 1 PORV. Hence, ideally

$$A \Rightarrow B \quad \text{is TRUE}$$

$$\equiv \neg A \vee B \quad \text{is TRUE}$$

hence.

$$\neg(\neg A \vee B) \quad \text{is UnSAT}$$

$$\equiv (A \wedge \neg B) \quad \text{is UnSAT}$$

We see that desired UnSAT is there in the computed UnSATCore and hence the implication relation holds good. Hence, $\{P, R\}$ remains to be a *Valid State Pair*. Now we consider the other state pair from the *specification* and *Implementation 2*. We supply the union of input PORVs i.e $\{w < 10\} \cup \{w < 5\}$ to MALL and get the *UnSATCore* $\neg A \wedge C$. Like the previous case, the desired *UnSAT* is $(A \wedge \neg C)$.

Algorithm 5 Algorithm for Symbolic Simulation Equivalence Checking

$Input$: Q_0^i, δ'^s, δ'^i, $ValidStatePairs$
$Output$: ($\mathcal{M}^s$, $\mathcal{M}^i$) are Simulation Equivalent

while $true$ **do**
  $ValidStatePairRenamed \leftarrow Rename_{x' \leftarrow x, y' \leftarrow y}(ValidStatePair)$
  $\delta^{global} \leftarrow \delta'^s \times \delta'^i$
  $GlobalNextStatesReachable \leftarrow \delta^{global} \times ValidStatePairRenamed$
  $NextStatesImpl \leftarrow ValidStatePair \times \delta'^i$
  $NextValidStatesImpl \leftarrow NextStatesImpl \times ValidStatePairRenamed$
  $\overline{NextStatesImplNotConsidered}$ $\leftarrow$ $NextValidStatesImpl$ $\times$
$\overline{GlobalNextStatesReachable}$
  $StatePairsTobeRemoved \leftarrow Smooth_{x',y'}\ \underline{(NextStatesImplNotConsidered)}$
  $RefinedStatePair \leftarrow ValidStatePair \times \overline{StatePairsTobeRemoved}$
  if $(ValidStatePair == RefinedStatePair)$ **then**
    if $(Q_0^i) \in RefinedStatePair$ **then**
      print *Specification simulates Implementation*
      break
    else
      print *Specification does not simulate Implementation*
      break
    end if
  else
    $ValidStatePair \leftarrow RefinedStatePair$
  end if
end while

But it is not contained in the computed $UnSATCore$ and hence the implication relation does not hold good. Hence, $\{P, T\}$ cannot be a *Valid State Pair.*

Following the calculation of initial $ValidStatePairs$, we use the symbolic BDD computation Algorithm 5 to refine $ValidStatePairs$ until we reach a *FixPoint.* If the *FixPoint* contains all the initial states of implementation $\mathcal{G}^i$ paired with at least one initial state of *specification* $\mathcal{G}^s$, then indeed specification simulates implementation.

To apply the algorithm, we follow the standard method to convert a labeled transition system to a Kripke Structure. We define the Kripke Structure corresponding to a LTS as follows:

Definition 4.2.1 *Kripke structure $\mathcal{M}$ equivalent to the controller $\mathcal{G} = \langle\ Q,\ I,\ \delta,\ Q_0,\ \mathcal{AP},\ var,\ \mathcal{L}\ \rangle$ can be defined as follows:*

$$\mathcal{M} = \langle\ Q',\ I',\ \delta',\ Q_0',\ \mathcal{AP}',\ var',\ \mathcal{L}'\ \rangle$$

where:

- $Q' = Q \times 2^I$. *We denote a state $q_i' \in Q'$ as a pair $\langle\ q_i, a_i\ \rangle$, where $q_i \in Q$ and $a_i \in 2^I$,*

- $Q'_0 = Q_0 \times 2^I$,
- $\delta' \subseteq Q' \times Q'$ *is the transition relation, such that* $(q'_i, q'_j) \in \delta'$ *iff* $(\underbrace{q_i, a_i}_{q'_i}, \underbrace{q_j, b_i}_{q'_j}) \in \delta$, $b_i \in 2^I$,
- $\mathcal{AP}' = \mathcal{AP} \cup I$,
- $\mathcal{L}' : Q' \rightarrow 2^{\mathcal{AP}}$ *is a labeling function, such that* $\mathcal{L}'(q'_i) = \mathcal{L}(q_i) \cup a_i$, *where* $q'_i = (q_i, a_i)$. ■

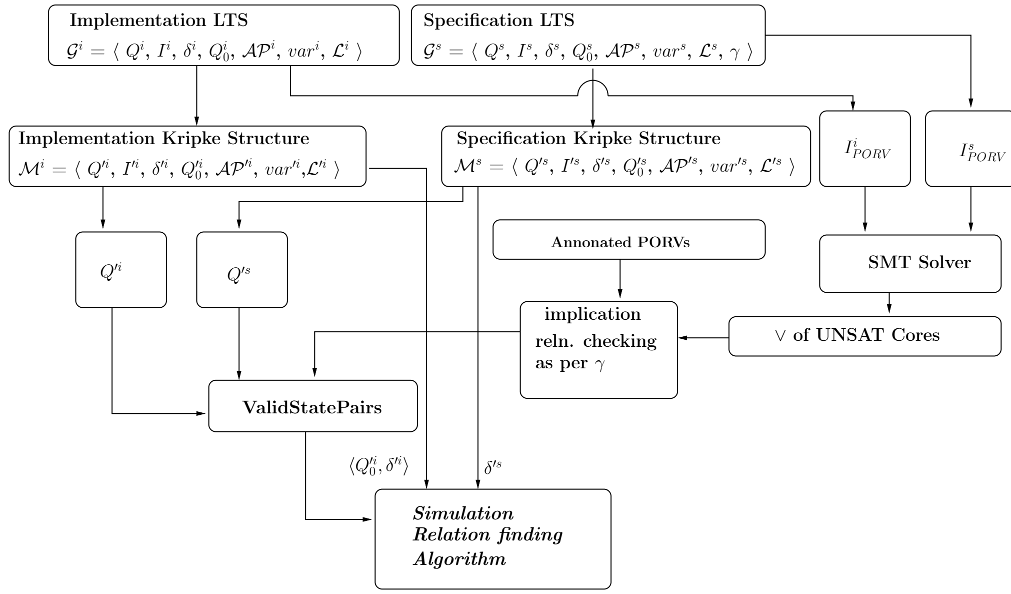


Figure 4.7: Simulation Relation Finding Tool Flow

We use $\mathcal{M}^s = \langle\ Q'^s,\ I'^s,\ \delta'^s,\ Q'^s_0,\ \mathcal{AP}'^s,\ var'^s,\ \mathcal{L}'^s\ \rangle$ for Kripke structure of the *specification LTS* $\mathcal{G}^s$ and $\mathcal{M}^i = \langle\ Q'^i,\ I'^i,\ \delta'^i,\ Q'^i_0,\ \mathcal{AP}'^i,\ var'^i,\ \mathcal{L}'^i\ \rangle$ for Kripke structure of the *implementation LTS* $\mathcal{G}^i$

For example, the Kripke structure equivalent of the *specification LTS* of Figure 4.1(a), can be expressed as the tuple $\mathcal{M}^s = \langle\ Q'^s,\ I'^s,\ \delta'^s,\ Q'^s_0,\ \mathcal{AP}'^s,\ var'^s,\ \mathcal{L}'^s\ \rangle$ where,

- $Q'^s = \{q_0, q_1\} \times 2^{\{w<10, w>85, w\geq 10, w\leq 85\}}$
- $Q'^s_0 = \{q_0\} \times 2^{\{w<10, w>85,, w\geq 10, w\leq 85\}}$
- $\mathcal{AP}'^s = \{Off,\ On,\ w < 10,\ w > 85, , w \geq 10, w \leq 85\}$
- $\mathcal{L}'^s(q'^s_i) =$

$$\begin{cases} \{Off\} \cup a^s_i, q^s_i = q_0 \\ \{On\} \cup a^s_i, q^s_i = q_1 \end{cases}$$

where $q_i'^s = (q_i^s, a_i^s)$ ■

The steps of the proposed symbolic BDD based *simulation relation* finding methodology is shown in Figure 4.7.

4.3. Concluding Remarks

In this chapter we have proposed a formal methodology to check *simulation relation* between predicate labeled transition systems. We have implemented a symbolic *simulation relation* finding algorithm by extending KS algorithm and have shown the necessary transformation steps.

Chapter 5
Feature based Equivalence Checking with AMS-VL over Simulation Trace

In Chapter 4, we studied the problem of checking simulation relations between a specification and implementation of digital controllers for hybrid systems. This methodology does not work for comparing the hybrid system as a whole, which consists of the controller and its analog environment. The demand for automating the comparison between analog and mixed signal (AMS) models has been growing in the circuit design community due to the ubiquitous use of analog components in large scale digital integrated circuits.

In the AMS domain, equivalence is typically defined in terms of specific *features*. In this chapter, we leverage the AMS-VL modules described in Chapter 3 to capture the features that form the basis of equivalence for a circuit family. The block diagram of Figure 5.1 illustrates the general idea behind our approach.

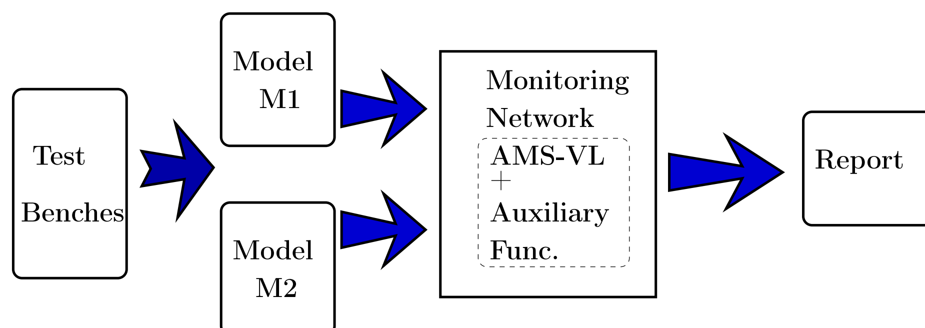


Figure 5.1: Conformance of Two AMS Models / Circuits

Two AMS circuits / models are simulated together with the help of the same test bench. The output of the circuits are either directly connected to the AMS-VL modules or to the auxiliary function modules. The combination of AMS-VL modules and the auxiliary modules form the conformance checking network. The

features based on which we want to check conformance may be a simple temporal behavior or may need to be calculated from the simulation trace with the help of auxiliary functions. The conformance report will be generated by the AMS-VL modules. In Section 5.1 we explain several topologies for finding conformance. In this chapter we use *equivalence* and *conformance* synonymously. In Section 5.2 we explain the topologies with suitable examples consisting of LDOs and Buck regulators.

5.1. Different topologies for Online Conformance

In this section we describe several topologies for online conformance checking based on the combination of the AMS-VL modules and auxiliary function modules.

1. **Topology-I** :

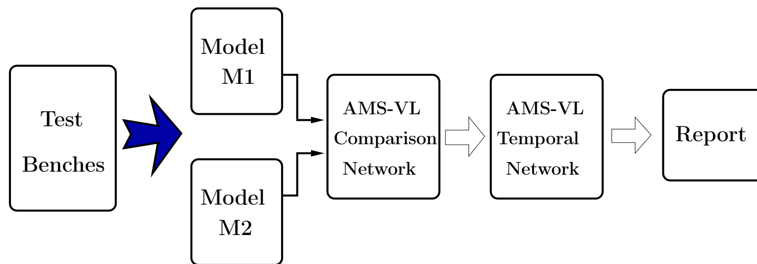


Figure 5.2: Block Diagram of Topology-I

This is the most simple topology for conformance checking. We assume that the two models of the AMS circuits are synchronized in time. In this case, the models / circuits are simulated with the same test bench and the signals of the output pins are compared in the *comparison network* to check if they are within certain tolerance limits specified by the user and further it is checked that the outputs remain within the tolerance limits over certain time period. The later part is checked in the *temporal network*. This kind of conformance checking network can only be used when the modes of operation of the two AMS models / circuits are in perfect sync. But this is a very restricted assumption and hence we refine this topology to Topology-II as shown in Figure 5.3.

2. **Topology-II** :

In most of the cases, it may happen that the two models / circuits are not synchronized in time and it may lead to improper comparisons of the output signals over time. Hence, it is necessary to initiate the comparison only when both of the AMS models / circuits are in same mode of operation. To do

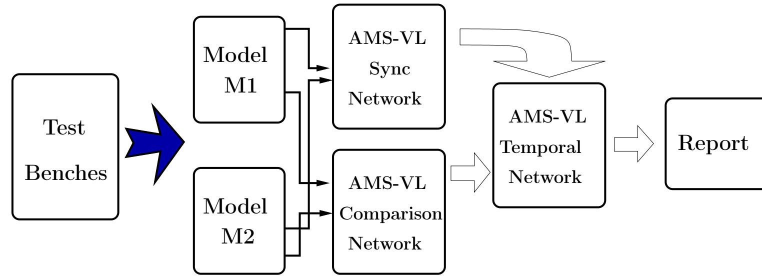


Figure 5.3: Block Diagram of Topology-II

that, we introduce a *sync network* along with the Topology-I of Figure 5.2. The sync network will trigger the *temporal network* to check whether indeed the temporal signals are within the tolerance limits over the desired time duration.

3. **Topology-III** :

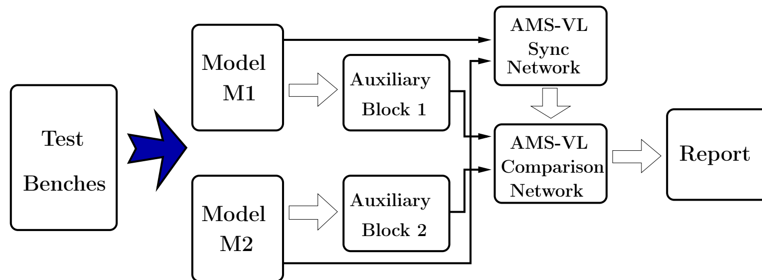


Figure 5.4: Block Diagram of Topology-III

In practical scenarios, it may be the case that the property with respect to which the conformance is desired, may not be a simple temporal property. Some properties may need to be calculated from the temporal trace with the help of the *auxiliary modules*, for example properties related to *rise time*, *settling time*, *frequency* etc. needs to be calculated from the temporal signal with some additional calculation on the temporal trace. For this we use this topology. As shown in the Figure 5.4, the *auxiliary blocks* will calculate the necessary features in the mode of interest. The sync network will trigger the comparison network only when both of the circuits are in the same mode of the operation so that the *comparison network* can compare between correct auxiliary features. It may be noted that in this topology there is no online comparison of temporal behaviors. Rather, we consider only those features which are non-temporal in nature, such as *rise time*, *settling time*, *dropout voltage* (of a LDO). These features are typically exhibited once during simulation and hence do not require a fully online comparison.

4. **Topology-IV** :

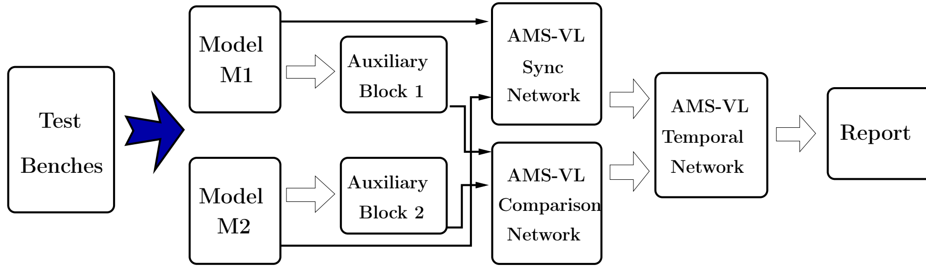


Figure 5.5: Block Diagram of Topology-IV

In this topology, we can check those auxiliary features that have to be checked over time. Here, the *sync network* triggers the temporal modules when both the AMS models / circuits are in same mode of operation and the feature has to be compared. This kind of topology is indeed required as explained in the Example 5.4.

5.2. Examples of Conformance Checking Networks

In this section we explain the above mentioned topologies with the help of the following examples.

5.2.1. Example of Topology-I

Consider the following example :

Example 5.1 *When a LDO enters its regulatory mode of operation, the maximum difference between the output voltage of the specification and the implementation LDOs should not exceed $\pm 0.05V$ for at least $20\mu s$. We assume that the two models enter the regulatory mode at the same time (which has to be enforced in the co-simulation environment).*

In this case, the feature is directly available from the temporal trace and hence needs no auxiliary calculation. Also, as the models enter the regulatory mode at the same time, no additional synchronization network is required. Hence, we use Topology-I to check conformance w.r.t this property.

In this example, the *ArithmeticOperator*, *PredicateEvaluators* and the *BoolOperator* forms the comparison network to check if the output voltage difference is within ± 0.05V. We use *EventdetectorDeglitched* (derived module as shown in Chapter 3, Example 3.1) to trigger the *GlobalOperator* module to check the temporal requirement. Since, the *EventdetectorDeglitched* compares the output of LDO_1 with a static value to trigger the *GlobalOperator*, hence we make it a part of the comparison network. *GlobalOperator* forms the temporal network.

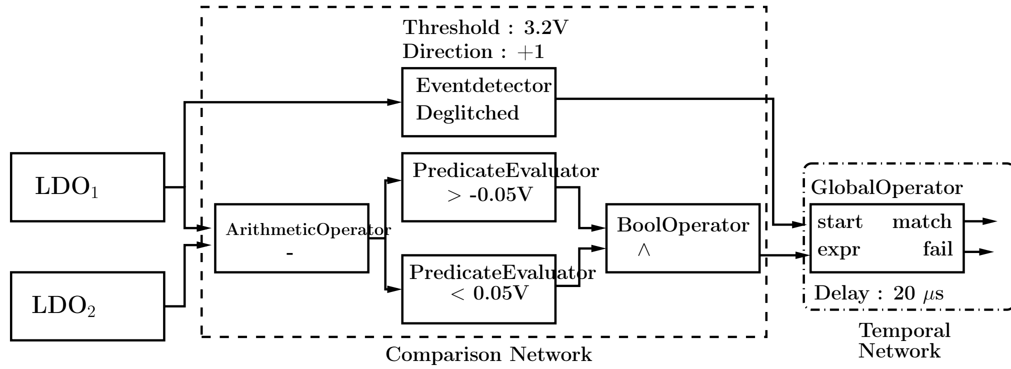


Figure 5.6: Monitoring Network for Example 5.1

5.2.2. Example of Topology-II

Consider the following example :

Example 5.2 *While in the startup mode of operation, the difference between the output voltages of the implementation and specification LDOs should remain within ±0.5V. We assume the normal steady state output voltage of the LDO is 3.5 Volts, and the startup mode is indicated when the output voltage remains within 5%-95% of steady state value.* ■

In this example the feature is directly available from the temporal simulation data and needs no auxiliary calculations. But we have to use a synchronization network to ensure that the comparison is done when the output voltages are within 5%-95% of its steady state value. Hence we use Topology-II to check the conformance.

As shown in the Figure 5.7, the *PredicateEvaluator*s and the *BoolOperator*s form the synchronization network to indicate that both the LDOs are in start-up mode of operation. The *ArithmeticOperator*, *PredicateEvaluator*s and the *BoolOperator* form the comparison network to check whether the difference of the output voltage is within ±0.5 V. The *PredicateAssert* forms the temporal network which generates the conformance report.

5.2.3. Example of Topology-III

Consider the following example :

Example 5.3 *While going from start-up mode to PFM mode of operation, the settling time of the implementation and the specification Buck regulator should not differ by ±1µs.* ■

Unlike the Example 5.2, the feature i.e. the settling time is not directly available from the temporal simulation data and hence need to be calculated with the

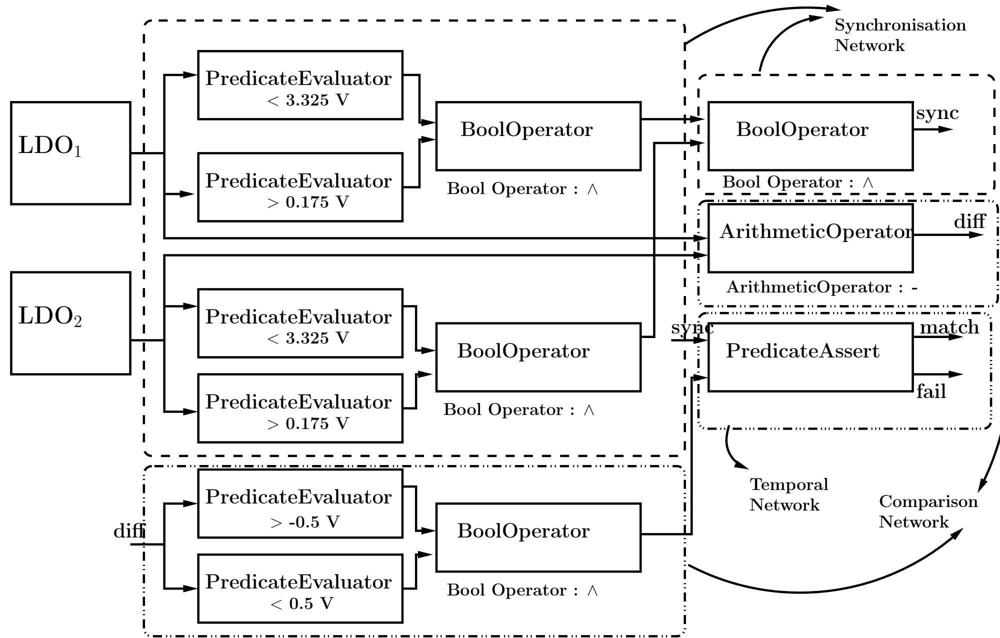


Figure 5.7: Monitoring Network for Example 5.2

help of the auxiliary modules. Also, we need a synchronization network to make sure that both the Buck regulators have entered into the PFM mode of operation from the startup mode. Also we need a comparison network to check if the *settling time* of the two Buck regulators are within the tolerance level or not. Hence, we use the Topology-III. The detailed conformance checking network is shown in the Figure 5.8.

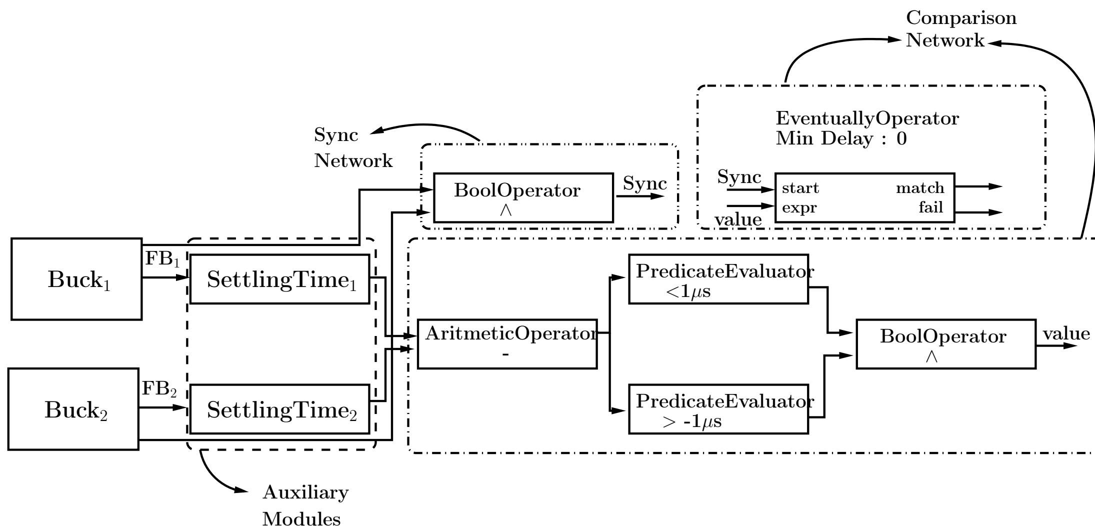


Figure 5.8: Monitoring Network for Example 5.3

As shown in the Figure 5.8, the *SettlingTime* modules will calculate the settling time of the output voltage at the *FB* pin and they constitute the auxiliary func-

tion network. The *ArithmeticOperator*, *PredicateEvaluators*, *BoolOperator* and the *EventuallyOperator* forms the comparison network. Here, *EventuallyOperator* does not work as a temporal module rather as soon as it receives the synchronization trigger from the synchronization network (formed by the single *BoolOperator*), it checks only for once whether the difference between settling time is indeed within the proposed band of tolerance. *EventuallyOperator* reports the conformance only once after the synchronization trigger is received.

5.2.4. Example of Topology-IV

Consider the following example :

Example 5.4 *In the PWM mode of operation, starting from beginning up to any time point, the maximum difference between the frequency of oscillation at the switching pin of specification and implementation Buck regulators should not exceed 500 Hz.* ■

In this example, frequency of oscillation is the feature of concern which is not available immediately from the temporal simulation data of Buck regulators and hence need to be calculated using some auxiliary modules. Further, it may be the case, that even if both the models / circuits are stimulated using the same testbench, they may not enter into the PWM mode of operation simultaneously. Hence, if we do not synchronize the conformance checking network, then a misleading report may be generated. Hence, we need auxiliary modules to calculate the *frequency* of oscillation of PWM mode and also a synchronization network which is available in the Topology-IV. We show the exact conformance checking network in Figure 5.9 and an implementation in Cadence Virtuoso Environment in Figure 5.10. We explain the network in detail in next paragraph.

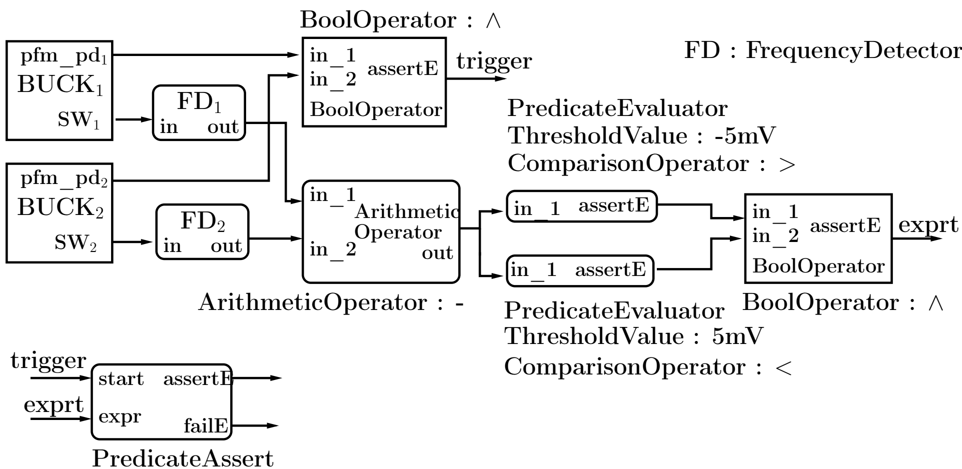


Figure 5.9: Monitoring Network for Example 5.4

Both the specification Buck regulator ($BUCK_1$ in the Figure 5.9) and the implementation Buck regulator ($BUCK_2$ in the Figure 5.9), are simulated by the identical testbench. Two *FrequencyDetector* auxiliary modules calculates the frequency of switching at the *SW* pins and converts the frequency of oscillation in an equivalent voltage output. When the signal *pfm_pd* is low, it indicates that the corresponding Buck regulator is in PWM mode of operation. When the both Buck regulators are in PWM mode of operation, the left most *BoolOperator* of Figure 5.9 asserts its output and keeps it asserted until one of the Buck comes out of the PWM mode. This *BoolOperator* is the required *sync network* as shown in Figure 5.5. The output of this *BoolOperator* triggers the *PredicateAssert* module which will check the conformance. The *ArithmeticOperator* calculates the difference of the frequency of the two Buck regulators. The two *PredicateEvaluator* modules check whether the difference of the frequency is within ±5 mV. The combination of two *PredicateEvaluators* and the *ArithmeticOperator* form the *comparison network* of Figure 5.5. If the difference is within the band of tolerance, the *BoolOperator* module in the right hand side will make its output asserted which will be the expression to check for the *PredicateAssert* module. Following the working principle of *PredicateAssert*, if the expression remain asserted till both the Buck regulators are in PWM mode of operation, then they conform with each other with respect to this property.

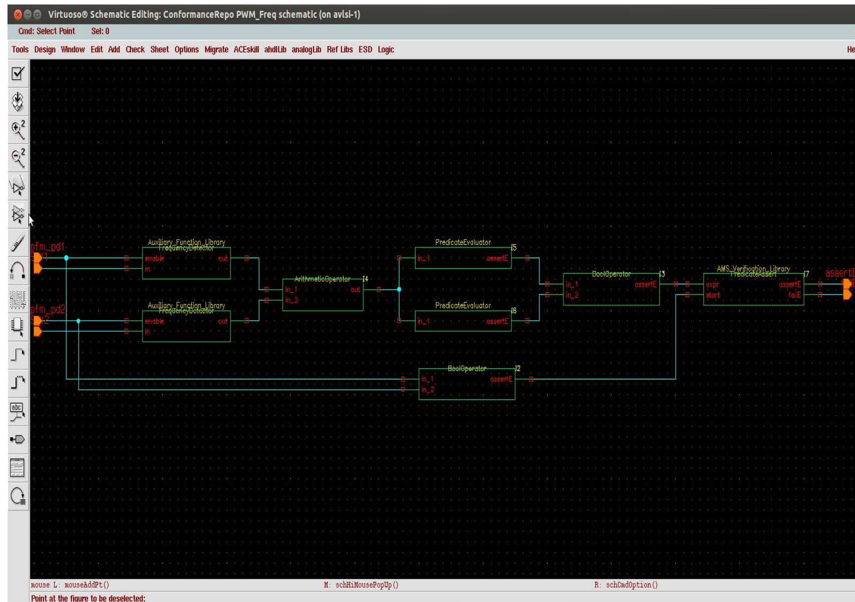


Figure 5.10: Schematic of Example 5.4

5.3. Concluding Remark

In this chapter we have shown several topologies to check conformance online between two AMS models / circuits. We have shown with suitable examples how the definition of the conformance between two AMS models / circuits can be translated to a conformance checking network using the AMS-VL modules and the auxiliary functions. As explained in Chapter 3, introduction of the additional library modules will increase the simulation overhead but we believe that the online conformance checking capability that the network of AMS-VL monitors provide will outweigh the simulation overhead.

Chapter 6
Conclusion

In this chapter we summarize the contributions of this work and discuss some possible directions for future research. The focus of this thesis has been on the verification of AMS circuits and digital controllers for AMS circuits / models. Towards this goal, we have developed a library of modules that can be graphically composed on a schematic to compose verification networks for different properties to be verified over the simulation trace of the AMS circuits. We have also leveraged this library to develop conformance checking monitors between two AMS circuits / models. We have also studied the verification of digital controllers for analog/hybrid systems, and have proposed a symbolic methodology to check conformance of digital controllers of AMS circuits. The following section summarizes our achievements.

6.1. Summary of Achievements

The thesis fulfills the following objectives:

1. **AMS Verification Library :** The problem of simulation trace based verification of AMS circuits is addressed by creating a library of passive online checker modules. The primary achievements include:

    - We have proposed a library of parameterized modules which can be connected graphically on a schematic to create verification networks for properties. We have demonstrated that this library can be easily imported into existing commercial industrial design and verification platforms, thereby significantly enhancing their verification capabilities.
    - We have shown that the library modules can be interfaced seamlessly with suitable auxiliary functions for verifying AMS properties that are not direct functions of time and require further calculation.

2. **Simulation Relation finding of LTSs :** The problem of finding simulation relation between LTS labeled with PORVs is addressed in this work. The primary achievements are as follows.

- We have proposed a formal level of abstraction for the *specification* and *implementation* controller.
- We have shown that the *refinement directives* on the transition guard conditions of specification controller can be used to account for the non-ideal situations encountered by the implementations and can help to ensure that the implementations never violates design intent in the strongest non-ideal environment.
- We show that the problem of checking trace-based simulation relation in such PORV-labeled LTSs reduces to the problem of checking a path-based simulation relation. This leads us to an efficient algorithm for checking the simulation relation leveraging the philosophy of the well known Kanellakis-Smolka algorithm.

3. **Feature based Conformance Checking of AMS Circuits :** In this work we have introduced a *feature based equivalence* checking approach for AMS circuits leveraging the online monitoring capability of AMS-VL modules. We have shown that with the help of AMS-VL modules and the auxiliary function modules, such notion of conformance can be translated into monitor networks which can check conformance over simulation runs and can generate the conformance report.

6.2. Future Work

The research work presented in thesis leaves several open directions for future research. We outline some of them briefly.

1. In our approach we have used Verilog-AMS to implement the modules in library. The arsenal of auxiliary functions that can be used to model circuit behaviors are mostly limited by the mathematical functional support of Verilog-AMS. As we know, Matlab has a wide support of different in-built auxiliary functions through its Simulink / Stateflow interface. These auxiliary functions can be used to check properties in various transform domains including time and frequency domains. It would be a good exercise to port the library developed in this thesis into the Matlab environment. The impact will be two fold; Firstly the in-built rich temporal and frequency domain functions can be used as auxiliary functions to model complex properties of AMS circuits. Secondly, with current integration of Matlab with Mixed-Mode simulators like Cadence®NCSIM, simulation dump monitoring and further processing can be easily done in Matlab. This will create a unified platform for devising new verification methodologies for AMS Circuits.

2. In digital domain, the environment of the controller is constrained with the help of *assume properties* (Chapter 17 of [6]) to deal with more realistic environments. In this work, we have dealt with digital controllers where the analog environment is unconstrained, that is, the analog plant is free to take all possible values for the real valued variables. But typically in a controlled environment, the dynamics of the analog plant is restricted such that the values of the real valued variables are constrained and this restriction can be modeled by an extension of the *assume properties* of the digital domain. It may be interesting to extend the present algorithm to find simulation relations under environment constraints modeled by *assume properties*.

To conclude this thesis, we feel that conformance checking is one of the main components in the design cycle of large AMS circuits. But till date, hardly any such methodology has been developed to check conformance between AMS circuits and their digital controllers which are controlled by PORVs. Thus a methodology was required to address this issue. We believe that this thesis has addressed some of the issues for finding conformance between AMS circuits and their controllers both formally and based on simulation traces in an effective way which in turn can eventually be applied for other large AMS circuits.

Appendix A
Sample Auxiliary Function Modules

In this chapter we present two *auxiliary function* modules which we used to verify different properties of LDO and BUCK with the help of AMS-VL modules. The first module was used to model startup of LDOs and BUCKs in the integrated PMU circuit. The other module was used to measure *duty cycle* in some properties of a Phase Locked Loop (PLL).

A.1. Auxiliary Module to model Start-Up of BUCK and LDOs

The following auxiliary function module models the startup behaviour of BUCKs and LDOs. It follows the equation $V_{out} = V_0 * (1 - e^{\frac{t}{\tau}})$, where V_0 is the voltage parameter depending upon the LDO and BUCK and τ is the time constatnt for that LDO and BUCK respectively. The module has several enable inputs like `LDO_1_EN`, `LDO_2_EN` (for LDOs) and `BUCK_ENABLE` (for BUCK) which depending upon proper signals from the circuit enable the calculation of corresponding *auxiliary functions*. The ideal values of the *auxiliary functions* are available at the output `LDO_1_AUX`, `LDO_2_AUX` etc.(for LDOs) and at `BUCK_AUX` for BUCK Regulator.

```
// Verilog-AMS HDL for "Auxiliary_Function_Library",
// "AuxiliaryFunctionBuckLDO"
// Written By Debjit Pal
// Indian Institute of Technology Kharagpur
// Dept. of Computer Science and Engineering

‘include "ams_verif_defines.h"
‘default_discipline logic
module AuxiliaryFunctionBuckLDO (BUCK_AUX, LDO_1_AUX, LDO_2_AUX,
```

```
LDO_3_AUX, LDO_4_AUX, LDO_5_AUX, LDO_6_AUX, BUCK_ENABLE,
LDO_1_EN, LDO_2_EN, LDO_3_EN, LDO_4_EN, LDO_5_EN, LDO_6_EN);
input BUCK_ENABLE, LDO_1_EN, LDO_2_EN, LDO_3_EN;
input LDO_4_EN, LDO_5_EN, LDO_6_EN;
output BUCK_AUX, LDO_1_AUX, LDO_2_AUX, LDO_3_AUX;
output LDO_4_AUX, LDO_5_AUX, LDO_6_AUX;
electrical BUCK_ENABLE, LDO_1_EN, LDO_2_EN, LDO_3_EN;
electrical LDO_4_EN, LDO_5_EN, LDO_6_EN;
electrical BUCK_AUX, LDO_1_AUX, LDO_2_AUX, LDO_3_AUX;
electrical LDO_4_AUX, LDO_5_AUX, LDO_6_AUX;
real save_buck_en_time;
real save_ldo_1_en_time;
real save_ldo_2_en_time;
real save_ldo_3_en_time;
real save_ldo_4_en_time;
real save_ldo_5_en_time;
real save_ldo_6_en_time;
integer state_buck;
integer state_ldo_1, state_ldo_2, state_ldo_3;
integer state_ldo_4, state_ldo_5, state_ldo_6;
parameter real ThresholdValue = `AMS_THRESHOLDVALUE_DEFAULT;
parameter real TimeTolerance  = `AMS_TIMETOLERANCE_DEFAULT;
parameter real ValueTolerance = `AMS_VALUETOLERANCE_DEFAULT;
initial begin
    save_buck_en_time  = 0.0;
    save_ldo_1_en_time = 0.0;
    save_ldo_2_en_time = 0.0;
    save_ldo_3_en_time = 0.0;
    save_ldo_4_en_time = 0.0;
    save_ldo_5_en_time = 0.0;
    save_ldo_6_en_time = 0.0;
end
always @(cross(V(BUCK_ENABLE) - ThresholdValue, +1, TimeTolerance,
  ValueTolerance)) begin
  state_buck = 1;
  save_buck_en_time = $abstime;
end
always @(cross(V(LDO_1_EN) - ThresholdValue, -1, TimeTolerance,
 ValueTolerance)) begin
  state_ldo_1 = 1;
  save_ldo_1_en_time = $abstime;
```

```
end
always @(cross(V(LDO_2_EN) - ThresholdValue, -1, TimeTolerance,
 ValueTolerance)) begin
  state_ldo_2 = 1;
  save_ldo_2_en_time = $abstime;
end
always @(cross(V(LDO_3_EN) - ThresholdValue, -1, TimeTolerance,
 ValueTolerance)) begin
  state_ldo_3 = 1;
  save_ldo_3_en_time = $abstime;
end
always @(cross(V(LDO_4_EN) - ThresholdValue, -1, TimeTolerance,
 ValueTolerance)) begin
  state_ldo_4 = 1;
  save_ldo_4_en_time = $abstime;
end
always @(cross(V(LDO_5_EN) - ThresholdValue, -1, TimeTolerance,
 ValueTolerance)) begin
  state_ldo_5 = 1;
  save_ldo_5_en_time = $abstime;
end
always @(cross(V(LDO_6_EN) - TimeTolerance, -1, TimeTolerance,
 ValueTolerance)) begin
  state_ldo_6 = 1;
  save_ldo_6_en_time = $abstime;
end
analog begin
@(initial_step)
begin
  V(BUCK_AUX)  <+ 0.0;
  V(LDO_1_AUX) <+ 0.0;
  V(LDO_2_AUX) <+ 0.0;
  V(LDO_3_AUX) <+ 0.0;
  V(LDO_4_AUX) <+ 0.0;
  V(LDO_5_AUX) <+ 0.0;
  V(LDO_6_AUX) <+ 0.0;
end
if(state_buck == 1)
      V(BUCK_AUX) <+ 0.5*(1 - exp((save_buck_en_time -
                     $abstime)/10e-6));
else
```

```
        V(BUCK_AUX) <+ 0.0;
  if(state_ldo_1 == 1)
        V(LDO_1_AUX) <+ 3.4*(1 - exp((save_ldo_1_en_time -
                      $abstime)/50e-6));
  else
        V(LDO_1_AUX) <+ 0.0;
  if(state_ldo_2 == 1)
        V(LDO_2_AUX) <+ 3.4*(1 - exp((save_ldo_2_en_time -
                      $abstime)/50e-6));
  else
        V(LDO_2_AUX) <+ 0.0;
  if(state_ldo_3 == 1)
        V(LDO_3_AUX) <+ 3.4*(1 - exp((save_ldo_3_en_time -
                      $abstime)/50e-6));
  else
        V(LDO_3_AUX) <+ 0.0;
  if(state_ldo_4 == 1)
        V(LDO_4_AUX) <+ 3.4*(1 - exp((save_ldo_4_en_time -
                      $abstime)/50e-6));
  else
        V(LDO_4_AUX) <+ 0.0;
  if(state_ldo_5 == 1)
        V(LDO_5_AUX) <+ 3.4*(1 - exp((save_ldo_5_en_time -
                      $abstime)/50e-6));
  else
        V(LDO_5_AUX) <+ 0.0;
  if(state_ldo_6 == 1)
        V(LDO_6_AUX) <+ 3.4*(1 - exp((save_ldo_6_en_time -
                      $abstime)/50e-6));
  else
        V(LDO_6_AUX) <+ 0.0;

  end
  endmodule
```

A.2. Auxiliary Module to measure Frquency of PLL

The following module calculates the *frequency* of an input signal at `in` port and produces output voltage at `out` port equal to the frequency value.

```
// Verilog-AMS HDL for "Auxiliary_Function_Library"
// "DutyCycleMeasure"
// Written By Debjit Pal
// Indian Institute of Technology Kharagpur
// Dept. of Computer Science and Engineering

‘include"disciplines.vams"
‘include"constants.vams"
‘timescale 1ns / 1ps
 module FrequencyDetector(enable, in, out);
 output out;
 input in, enable;
 logic in, enable;
 electrical out;
 real Vout;
 integer oddEvenCount;
 real saveTime1 = 0.0, saveTime2 = 0.0, timePeriod = 0.0;
 always@(posedge in) begin
 if(oddEvenCount == 1) begin
    oddEvenCount = 0;
    saveTime1 = $abstime;
    timePeriod = (saveTime1 > saveTime2) ? saveTime1
                 - saveTime2 : saveTime2 - saveTime1;
 end
 else if(oddEvenCount == 0) begin
    oddEvenCount = 1;
    saveTime2 = $abstime;
    timePeriod = (saveTime1 > saveTime2) ? saveTime1
                 - saveTime2 : saveTime2 - saveTime1;
 end
 else begin
    oddEvenCount = 0;
    saveTime1 = $abstime;
    saveTime2 = $abstime;
    timePeriod = (saveTime1 > saveTime2) ? saveTime1
                 - saveTime2 : saveTime2 - saveTime1;
 end
 if(timePeriod > 0)
    Vout = 1/(timePeriod*1e9);
 else
    Vout = 0.0;
```

```
  end
  analog begin
    if(enable)
       V(out) <+ Vout;
    else
       V(out) <+ 0.0;
  end
endmodule
```

Appendix B
Testcases and Sample Properties of AMS-VL

B.1. Low Dropout Regulator (LDO)

A low dropout (LDO) regulator is a linear DC voltage regulator that can operate with a very small input-output differential voltage. They are mainly used to provide regulated output voltage. Figure B.1 shows a simplified block diagram of a typical LDO regulator circuit. The operational modes of this LDO regulator are as follows:

- **Shutdown Mode:** If enable is not asserted, or either the bias current, or the input supply voltage are not within the specified range, then the LDO regulator remains in shutdown mode.
- **Start-up Mode:** When all the enables, bias current, and input supply voltage are within the specified range, the LDO regulator enters into the start-up mode of operation.
- **Regulatory Mode:** When the output voltage reaches the desired steady state value is the start-up mode, the LDO regulator switches to the regulatory mode. In this mode of operation steady output voltage is maintained.
- **Dropout Mode:** For proper functionality of the LDO regulator, the relation:

  $$V_{in} > V_{out} + V_{min_drop}$$

  should hold, where V_{min_drop} is the minimum dropout voltage. If the input voltage starts falling, then the output voltage will remain constant as long as the above relation holds. But if the input-output differential voltage falls below the dropout value, then the output voltage starts falling below its rated value to keep the input-output difference above the dropout value. This mode of the LDO regulator, when output voltage falls with the fall of input voltage, is called the dropout mode of operation.

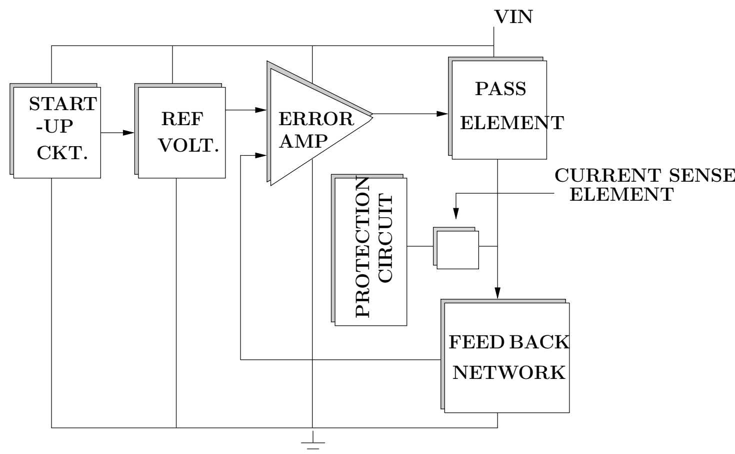


Figure B.1: Block Diagram of an LDO Regulator Circuit [59].

- **Short Circuit Mode:** When the output current crosses a certain current limit, the LDO regulator enters into the short circuit mode of operation.

The output voltage of the LDO regulator in different modes of operation has been shown in Figure B.2.

B.2. Voltage Mode Controlled BUCK Regulator

A buck regulator is a step-down DC-DC converters. Buck regulators are used to step down a higher level, unregulated input voltage to a regulated output voltage. Figure B.3 shows a simplified block diagram of a typical buck regulator circuit with the following operational modes.

- **Shutdown Mode:** If enable is de-asserted, or either the bias current, or the input supply voltage are not within the desired range, then the buck regulator remains in the shutdown mode of operation.

- **PWM Mode:** When all the enabling conditions are asserted with the bias current, and the input supply voltage within the specified ranges, then the output voltage of the buck regulator starts rising. Thereby it enters into the pulse width modulation (PWM) [32] mode of operation through the start-up phase.

- **PFM Mode:** At very light loads, the regulator enters the pulse frequency modulation (PFM) mode, and operates with reduced switching frequency and quiescent current to maintain high efficiency. During the PFM mode of operation, the regulator's output voltage is slightly higher than the nominal output voltage of the PWM mode of operation. There are two sub-categories of the PFM mode of operation. They are:

  - *PFM_rise Mode:* In this mode, the output voltage rises from the lower voltage limit to the upper voltage limit of the PFM mode. The pMOS switch

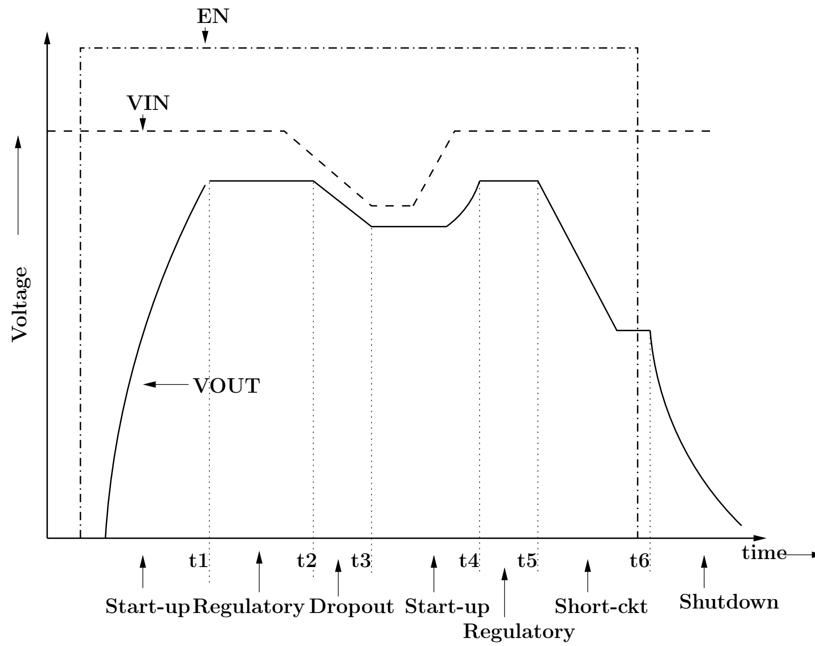


Figure B.2: Output Voltage of LDO Regulator in Different Modes of Operation.

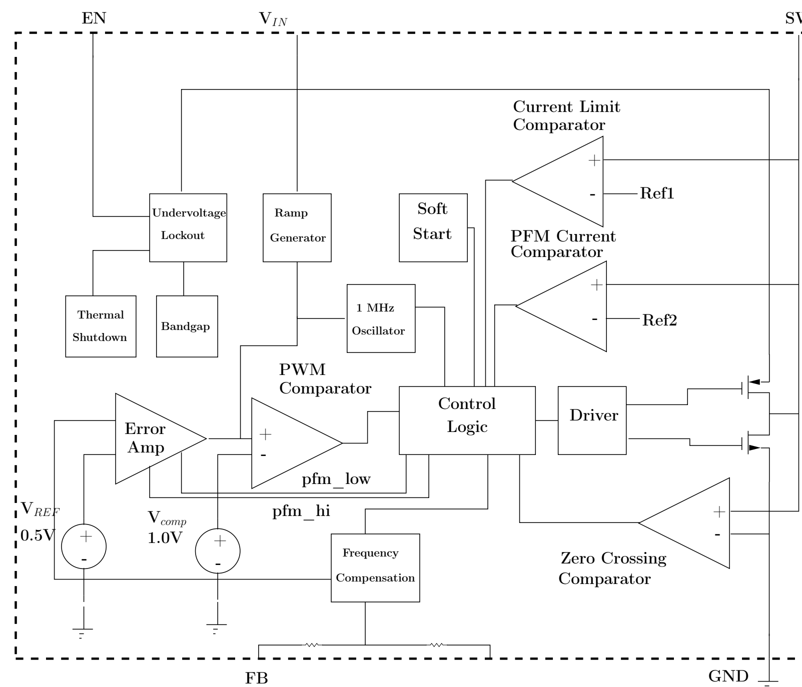


Figure B.3: Block Diagram of a Buck Regulator Circuit [47].

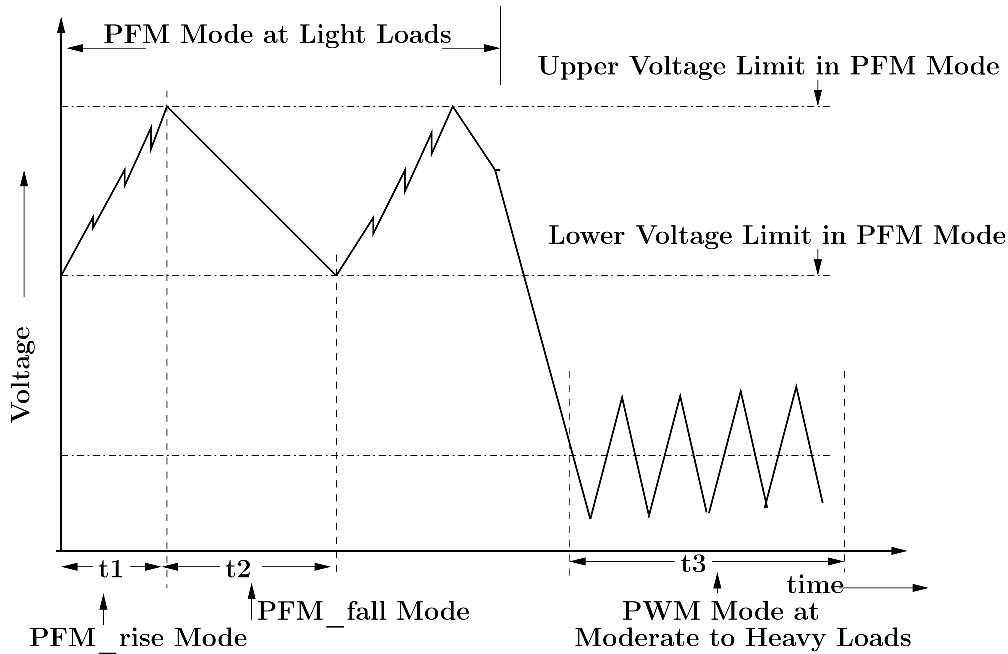


Figure B.4: Output Voltage of Buck Regulator Circuit [47].

remains on and nMOS switch remains off until the inductor current attains its peak value. Similarly, the nMOS switch remains on and the pMOS switch remains off until the inductor current ramps to zero.

– *PFM_fall Mode:* In this mode the output voltage falls from the upper voltage limit to the lower voltage limit of the PFM mode with both the pMOS and the nMOS switches at off positions.

The output voltage of a buck regulator in different modes of operation has been shown in Figure B.4 [47].

B.3. Sample Properties for Low Dropout Regulators

We verify the following set of properties on an Integrated network of Low Dropout Regulators (LDO) consisting of 6 LDOs. The inter-connection network has a well defined switching sequence. LDO-1 will startup automatically when the circuit has been excited with proper test bench. LDO-1 will trigger LDO-2 and in turn LDO-2 will trigger LDO-3. LDO-4 will be triggered by LDO-3. LDO-5 will be triggered by proper startup of LDO-1 and LDO-2 and LDO-6 will get its trigger if LDO-3, LDO-4 and LDO-5 have started properly.

Property B.3.1 *If LDO-1 is enabled then it will start within next 20 μs.*

Property B.3.2 *If LDO-1 enters in the startup mode, then within next 80 μs LDO-2 will start up.*

Property B.3.3 *If LDO-2 enters in the startup mode, then within next 100 μs LDO-3 will start up.*

Property B.3.4 *If LDO-3 enters in the startup mode, then within next 100 μs LDO-4 will start up.*

Property B.3.5 *If LDO-1 and LDO-2 have entered in the startup mode in proper sequence, then LDO-5 will enter startup mode eventually.*

Property B.3.6 *If LDO-3, LDO-4 and LDO-5 have entered startup mode in proper sequence, then LDO-6 will enter startup mode eventually.*

Property B.3.7 *If LDO-1 is low enabled, then within next 15 μs LDO-1 will cross 0.3V.*

Property B.3.8 *If LDO-2 is low enabled, then within next 20 μs LDO-2 will cross 0.3V.*

Property B.3.9 *If LDO-3 is low enabled, then within next 18 μs LDO-3 will cross 0.3V.*

Property B.3.10 *If LDO-4 is low enabled, then within next 15 μs LDO-4 will enter startup mode.*

Property B.3.11 *If LDO-5 is low enabled, then within next 15 μs, LDO-5 will enter startup mode.*

Property B.3.12 *If LDO-6 is low enabled, then within next 15 μs, LDO-6 will enter startup mode.*

Property B.3.13 *After LDO-1 enters startup mode, it will follow the auxiliary function specified by V(LDO_1_AUX) for next 430 μs with a tolerance value of 0.57V.*

Property B.3.14 *After LDO-2 enters startup mode, it will follow the auxiliary function specified by V(LDO_2_AUX) for next 430 μs with a tolerance value of 0.57V.*

Property B.3.15 *After LDO-3 enters startup mode, it will follow the auxiliary function specified by V(LDO_3_AUX) for next 350 μs with a tolerance value of 0.6V.*

Property B.3.16 *After LDO-4 enters startup mode, it will follow the auxiliary function specified by V(LDO_4_AUX) for next 350 μs with a tolerance value of 0.6V.*

Property B.3.17 *After LDO-5 enters startup mode, it will follow the auxiliary function specified by V(LDO_5_AUX) for next 350 μs with a tolerance value of 0.6V.*

Property B.3.18 *After LDO-6 enters startup mode, it will follow the auxiliary function specified by V(LDO_6_AUX) for next 300 μs with a tolerance value of 0.6V.*

Property B.3.19 *Within 300 μs of entering in the startup mode, LDO-1 will enter steady state mode.*

Property B.3.20 *Within 280 μs of entering in the startup mode, LDO-2 will enter steady state mode.*

Property B.3.21 *Within 300 μs of entering in the startup mode, LDO-3 will enter steady state mode.*

Property B.3.22 *Within 250 μs of entering in the startup mode, LDO-4 will enter steady state mode.*

Property B.3.23 *Within 250 μs of entering in the startup mode, LDO-5 will enter steady state mode.*

Property B.3.24 *Within 250 μs of entering in the startup mode, LDO-6 will enter steady state mode.*

Property B.3.25 *After* $V(out)_{LDO_1}$ *crosses 3.3V and enters steady state, the voltage will remain within 3.3V and 3.4V for next 5 μs and will check it forever.*

Property B.3.26 *After* $V(out)_{LDO_2}$ *crosses 3.3V and enters steady state, the voltage will remain within 3.3V and 3.4V for next 10 μs and will check it forever.*

Property B.3.27 *230 μs after entering into startup mode, LDO-3 will enter into the steady-state mode and* $V(out)_{LDO_3}$ *will remain within 3.3V and 3.4V for next 220 μs.*

Property B.3.28 *235 μs after entering into startup mode, LDO-5 will enter into the steady-state mode and* $V(out)_{LDO_5}$ *will remain within 3.3V and 3.4V for next 200 μs.*

B.4. Sample Properties for BUCK Regulators

We verify the following set of properties on an Integrated network of BUCK Regulators consisting of 2 BUCK Regulators. The interconnected network has a well defined switching sequence. BUCK-1 will startup initially when the circuit is excited with proper test bench. BUCK-2 will get triggered after BUCK-1 has entered startup mode.

Property B.4.1 *After BUCK-1 gets enabled, within 60 μs, it will enter startup mode.*

Property B.4.2 *After BUCK-2 gets enabled, within 60 μs, it will enter startup mode.*

Property B.4.3 *After BUCK-1 enters startup mode,* $V(FB)_{BUCK_1}$ *will follow the auxiliary function specified by V(BUCK_1_AUX) for 200 µs with a tolerance value of 0.3V.*

Property B.4.4 *After BUCK-2 enters startup mode,* $V(FB)_{BUCK_2}$ *will follow the auxiliary function specified by V(BUCK_2_AUX) for 200 µs with a tolerance value of 0.35V.*

Property B.4.5 *After BUCK-1 enters PWM mode, the frequency of oscillation will become 2.0MHz with a tolerance of 0.1MHz within next 20 µs and will check it forever at a regular interval of 10 µs.*

Property B.4.6 *After BUCK-1 enters PWM mode, the duty cycle of switching will become 0.0053 with a tolerance of 0.0001 within next 20 µs and will check it forever at a regular interval of 10 µs.*

Property B.4.7 *After BUCK-1 enters startup mode, the frequency of oscillation of BUCK-2 will become 2.0MHz with a tolerance of 0.1MHz within next 130 µs and will check it forever at a regular interval of 10 µs.*

Property B.4.8 *After BUCK-1 enters startup mode, the duty cycle of switching of BUCK-2 will become 0.0053 with a tolerance of 0.0001 within next 130 µs and will check it forever at a regular interval of 10 µs.*

Property B.4.9 *After 190 µs of entering startup mode, BUCK-1 will enter in steady state mode and its steady state voltage will remain within 0.5V and 0.56V for next 10 µs and will check it forever at a regular interval of 10 µs.*

Property B.4.10 *290 µs after BUCK-1 enters startup mode, BUCK-2 will enter in steady state mode and its steady state voltage will remain within 0.5V and 0.56V for next 10 µs and will check it forever at a regular interval of 10 µs.*

B.5. Sample Properties for Integrated Power Management Unit

We verify the following set of properties on an Integrated Power Management Unit (PMU) consisting of 1 BUCK Regulator and 4 Low Dropout Regulators (LDO). The PMU has a well defined switching sequence. BUCK will startup after the circuit has been excited with the proper test bench. BUCK-1 will trigger LDO-1 and in turn LDO-1 will trigger LDO-2. LDO-2 will trigger LDO-3. For successful triggering of LDO-4, both BUCK-1 and LDO-2 should startup properly.

Property B.5.1 *If the BUCK Regulator is a enabled then within next 140 μs the output of the BUCK Regulator will be above 0.3V.*

Property B.5.2 *If the BUCK regulator enters startup mode then within 160 μs the LDO-1 enters the startup mode.*

Property B.5.3 *If the LDO-1 enters the startup mode then within 100 μs the LDO-2 enters startup mode.*

Property B.5.4 *If the LDO-2 enters the startup mode then within 100 μs the LDO-3 enters startup mode.*

Property B.5.5 *If BUCK and LDO-2 enters startup then eventually LDO-4 will startup too.*

Property B.5.6 *If the BUCK Regulator enters startup mode then output voltage will follow the auxiliary function V(buck_ aux) for 200 μs with a tolerance of 0.5V.*

Property B.5.7 *If the LDO-1 enters startup mode then output voltage of LDO-1 will follow the auxiliary function described by V(ldo1_ aux) with a tolerance of 0.6 V for 360 μs.*

Property B.5.8 *If the LDO-2 enters startup mode then output voltage of LDO-2 will follow the auxiliary function specified by V(ldo2_ aux) with a tolerance of 0.6 V for 360 μs.*

Property B.5.9 *If the LDO-3 enters startup mode then output voltage of LDO-3 will follow the auxiliary function described by V(ldo3_ aux) with a tolerance of 0.6V for 280 μs.*

Property B.5.10 *If the LDO-4 enters startup mode then output voltage of LDO-4 will follow the auxiliary function described by V(ldo4_ aux) with a tolerance of 0.6V for 280 μs.*

Property B.5.11 *If the LDO-1 enters the startup mode then within 340 μs it will be in the steady state mode.*

Property B.5.12 *If the LDO-2 enters the startup mode then within 340 μs it will be in the steady state mode.*

Property B.5.13 *If the LDO-3 enters the startup mode then within 260 μs it will be in the steady state mode.*

Property B.5.14 *If the LDO-4 enters the startup mode then within 260 μs it will be in the steady state mode.*

Property B.5.15 *If LDO-1 is low-enabled then within next 20 μs it should start up.*

Property B.5.16 *If LDO-2 is low-enabled then within next 20 μs it should start up.*

Property B.5.17 *If LDO-3 is low-enabled then within next 20 μs it should start up.*

Property B.5.18 *If LDO-4 is low-enabled then within next 20 μs it should start up.*

Property B.5.19 *If LDO-1 is at steady state mode then the output voltage will be around 3.4V with a tolerance of 0.05V.*

Property B.5.20 *If LDO-2 is at steady state mode then the output voltage will be around 3.4V with a tolerance of 0.05V.*

Property B.5.21 *If LDO-3 is at steady state mode then the output voltage will be around 3.4V with a tolerance of 0.05V.*

Property B.5.22 *If LDO-4 is at steady state mode then the output voltage will be around 3.4V with a tolerance of 0.05V.*

Property B.5.23 *If LDO-1 is low-enabled then within next 360 μs it should be at steady state.*

Property B.5.24 *If LDO-2 is low-enabled then within next 360 μs it should be at steady state.*

Property B.5.25 *If LDO-3 is low-enabled then within next 280 μs it should be at steady state.*

Property B.5.26 *If LDO-4 is low-enabled then within next 280 μs it should be at steady state.*

Property B.5.27 *After BUCK enters startup mode, the output voltage will be 0.5V with a tolerance of 0.07V.*

Property B.5.28 *After 350μs of LDO-1 enters startup mode $V(out)_{LDO1}$ will remain within 3.3V and 3.4V for next 80μs.*

Property B.5.29 *After 325μs of LDO-2 enters startup, $V(out)_{LDO2}$ will remain within 3.3V and 3.5V for next 40μs.*

Property B.5.30 *After 250μs of LDO-3 enters startup, $V(out)_{LDO3}$ will remain within 3.3V and 3.5V for next 30μs.*

Property B.5.31 *After 250μs of LDO-3 enters startup, $V(out)_{LDO3}$ will remain within 3.3V and 3.5V for next 30μs.*

Property B.5.32 *At PWM mode, switching frequency of BUCK Regulator will become 2MHz eventually with a tolerance of 0.1MHz.*

Property B.5.33 *At PWM mode, the duty ratio will eventually become 0.2.*

Bibliography

[1] Accellera. http://www.accellera.org/.

[2] Accellera Open Verification Library. http://www.accellera.org/activities/ovl.

[3] Accellera Verilog-AMS Language Reference Manual Analog and Mixed Signal Extensions to Verilog-HDL, version 2.4 edition (November 2006). http://www.accellera.org/downloads/standards/v-ams.

[4] Cadence AMS Simulator
. http://www.vtvt.ece.vt.edu/vlsidesign/tutorialmixedsignal_intro.php.

[5] Cadence NCSIM. http://www.cadence.com/products/cic/ams_designer.

[6] IEEE Std 1800-2009, "IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language, IEEE, 2010.".

[7] IEEE Std 1850-2010, IEEE Standard for Property Specification Languages (PSL), IEEE, 2010.

[8] MetiTarski Theorem Prover. http://www.cl.cam.ac.uk/~lp15/papers/Arith/.

[9] Open Vera Assertions. http://www.open-vera.com/.

[10] PHAver. http://www-verimag.imag.fr/~frehse/phaver_web/.

[11] PVS Theorem Prover. http://pvs.csl.sri.com/.

[12] VHDL-AMS. http://www.eda.org/twiki/bin/view.cgi/P10761/WebHome.

[13] Aceto, L., Ingolfsdottir, A., and Srba, J. *The Algorithmics of Bisimilarity.* Cambridge University Press.

[14] Althoff, M., Yaldiz, S., Rajhans, A., Li, X., Krogh, B. H., and Pileggi, L. T. Formal Verification of Phase-Locked Loops using Reachability Analysis and Continuization. In *2011 IEEE/ACM International Conference on Computer-Aided Design, San Jose, California, USA, November 7-10, 2011*, IEEE, pp. 659–666.

[15] Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T. A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science 138*, 1 (1995), 3–34.

[16] Alur, R., Courcoubetis, C., Henzinger, T. A., and Ho, P.-H. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In *Hybrid Systems* (1992), vol. 736 of *Lecture Notes in Computer Science*, Springer, pp. 209–229.

[17] Alur, R., Feder, T., and Henzinger, T. A. The Benefits of Relaxing Punctuality. *Journal of ACM 43*, 1 (1996), 116–146.

[18] Alur, R., and Henzinger, T. A. Real-Time Logics: Complexity and Expressiveness. *Information and Computation 104*, 1 (1993), 390–401.

[19] Alur, R., and Henzinger, T. A. A Really Temporal Logic. *Journal of ACM 41*, 1 (1994), 181–204.

[20] Baier, C., and Katoen, J.-P. *Principles of Model Checking.* MIT Press, 2008.

[21] Balivada, A., Hoskote, Y. V., and Abraham, J. A. Verification of Transient Response of Linear Analog Circuits. In *IEEE VLSI Test Symposium* (April 30 - May 3,1995, Princeton, New Jersey, USA), IEEE Computer Society, pp. 42–47.

[22] Bouali, A., and de Simone, R. Symbolic Bisimulation Minimisation. In *Computer Aided Verification* (Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1,1992), G. von Bochmann and D. K. Probst, Eds., vol. 663 of *Lecture Notes in Computer Science*, Springer, pp. 96–108.

[23] Browne, M. C., Clarke, E. M., Dill, D. L., and Mishra, B. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Transaction on Computers 35*, 12 (1986), 1035–1044.

[24] Browne, M. C., Clarke, E. M., and Grumberg, O. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science 59* (1988), 115–131.

[25] Bryant, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers 35*, 8 (1986), 677–691.

[26] Burch, J. R., Clarke, E. M., Long, D. E., Mcmillan, K. L., and Dill, D. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13* (1993), 401–424.

[27] Clarke, E. M., Donzé, A., and Legay, A. Statistical Model Checking of Mixed-Analog Circuits with an Application to a Third Order Delta-Sigma Modulator. In *Hardware and Software: Verification and Testing, 4th International Haifa Verification Conference, HVC 2008, Haifa, Israel, October 27-30, 2008. Proceedings* (2009), vol. 5394 of *Lecture Notes in Computer Science*, Springer, pp. 149–163.

[28] Dang, T., Donzé, A., and Maler, O. Verification of Analog and Mixed-Signal Circuits Using Hybrid System Techniques. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, vol. 3312 of *Lecture Notes in Computer Science*, Springer, pp. 21–36.

[29] Dang, T., and Nahhal, T. Randomized Simulation of Hybrid Systems For Circuit Validation. In *Forum on specification and Design Languages, FDL 2006, September 19-22, 2006, Darmstadt, Germany, Proceedings*, ECSI, pp. 9–15.

[30] Dastidar, T. R., and Chakrabarti, P. P. A Verification System for Transient Response of Analog Circuits. *ACM Transaction on Design Automation and Electronic System 12*, 3 (2007).

[31] Denman, W., Akbarpour, B., Tahar, S., Zaki, M. H., and Paulson, L. C. Formal Verification of Analog Designs using MetiTarski. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, IEEE, pp. 93–100.

[32] Erickson, R., and Maksimovic, D. *Fundamentals of Power Electronics*. Kluwer Academic Publishers, 2001.

[33] Frehse, G., Krogh, B. H., and Rutenbar, R. A. Verifying Analog Oscillator Circuits using Forward/Backward Abstraction Refinement. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March 6-10, 2006*, European Design and Automation Association, Leuven, Belgium, pp. 257–262.

[34] Frehse, G., Krogh, B. H., Rutenbar, R. A., and Maler, O. Time Domain Verification of Oscillator Circuit Properties. *Electr. Notes Theor. Comput. Sci. 153*, 3 (2006), 9–22.

[35] Ghosh, A., and Vemuri, R. Formal Verification of Synthesized Analog Designs. In *ICCD* (1999), pp. 40–45.

[36] Gupta, S., Krogh, B. H., and Rutenbar, R. A. Towards Formal Verification of Analog Designs. In *ICCAD* (November 7-11, 2004, San Jose, CA, USA), IEEE Computer Society / ACM, pp. 210–217.

[37] Hartong, W., Klausen, R., and Hedrich, L. Formal Verification for Nonlinear Analog Systems: Approaches to Model and Equivalence Checking. In *Advanced Formal Verification*. Springer, 2004, pp. 205–245.

[38] Hedrich, L., and Barke, E. A Formal Approach to Verification of Linear Analog Circuits with Parameter Tolerances. In *DATE* (Le Palais des Congrès de Paris, Paris, France,February 23-26, 1998), IEEE Computer Society, pp. 649–654.

[39] Hedrich, L., and Barke, E. A Formal Approach to Nonlinear Analog Circuit Verification. In *ICCAD* (San Jose, California, USA, November 5-9,1995), R. L. Rudell, Ed., IEEE Computer Society, pp. 123–127.

[40] Henzinger, M. R., Henzinger, T. A., and Kopke, P. W. Computing Simulations on Finite and Infinite Graphs. In *Foundations of Computer Science* (1995), pp. 453–462.

[41] Henzinger, T. A. The Theory of Hybrid Automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science* (New Brunswick, New Jersey, USA, July 27-30, 1996), IEEE Computer Society, pp. 278–292.

[42] Henzinger, T. A., Kopke, P. W., Puri, A., and Varaiya, P. What's Decidable about Hybrid Automata? *Journal of Computer and System Sciences 57*, 1 (1998), 94–124.

[43] Horowitz, M., Jeeradit, M., Lau, F., Liao, S., Lim, B., and Mao, J. Fortifying Analog Models with Equivalence Checking and Coverage Analysis. In *47th Design Automation Conference,DAC 2010, Anaheim, California, USA,* (July 13-18,2010), S. S. Sapatnekar, Ed., ACM, pp. 425–430.

[44] Kanellakis, P. C., and Smolka, S. A. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation 86*, 1 (1990), 43–68.

[45] Kanellakis, P. C., and Smolka, S. A. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada* (August 17-19, 1983), pp. 228–240.

[46] Little, S., and Mayers, C. Abstract Modeling and Simulation Aided Verification of Analog/Mixed-Signal Circuits. In *Formal Verification of Analog Circuits* (2008).

[47] LM3670. A DC-DC converter from National Semiconductor. http://www.national.com/ds/LM/LM3670.pdf.

[48] Maler, O., and Nickovic, D. Monitoring Temporal Properties of Continuous Signals. In *Joint International Conferences on Formal Modelling and Analysis of Timed Systems, (FORMATS) Formal Techniques in Real-Time and Fault-Tolerant Systems, (FTRTFT), Grenoble, France, Proceedings* (September 22-24, 2004), vol. 3253 of *Lecture Notes in Computer Science*, Springer, pp. 152–166.

[49] Maler, O., Nickovic, D., and Pnueli, A. Checking Temporal Properties of Discrete, Timed and Continuous Behaviors. In *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday* (2008), vol. 4800 of *Lecture Notes in Computer Science*, Springer, pp. 475–505.

[50] Mukherjee, S., and Dasgupta, P. Auxiliary State Machines and Auxiliary Functions: Constructs for Extending AMS Assertions. In *24th International Conference on VLSI Design, IIT Madras, Chennai, India* (2-7 January 2011), IEEE, pp. 52–57.

[51] Mukherjee, S., Dasgupta, P., and Mukhopadhyay, S. Auxiliary Specifications for Context-Sensitive Monitoring of AMS Assertions. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems 30*, 10 (2011), 1446–1457.

[52] Mukherjee, S., Dasgupta, P., Mukhopadhyay, S., Little, S., Havlicheck, J., and Chandrasekaran, S. Synchronizing AMS Assertions with AMS Simulation : From Theory to Practice. *ACM Transaction on Design Automation and Electronic System Accepted for Publication* (2012).

[53] Mukhopadhyay, R., Panda, S. K., Dasgupta, P., and Gough, J. Instrumenting AMS assertion Verification on Commercial Platforms. *ACM Transaction on Design Automation and Electronic System 14*, 2 (2009).

[54] Narayanan, R., Akbarpour, B., Zaki, M. H., Tahar, S., and Paulson, L. C. Formal verification of analog circuits in the presence of noise and process variation. In *DATE* (Dresden,Germany, March 8-12, 2010), IEEE, pp. 1309–1312.

[55] Nickovic, D. *Checking Timed and Hybrid Poperties : Theory and Applications.* PhD thesis, University of Joseph Fourier, 2008.

[56] Paige, R., and Tarjan, R. E. Three Partition Refinement Algorithms. *Society for Industrial and Applied Mathematics (SIAM) Journal on Computing 16*, 6 (1987), 973–989.

[57] Pnueli, A. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA* (, 31 October - 1 November 1977), IEEE Computer Society, pp. 46–57.

[58] Ranzato, F., and Tapparo, F. A New Efficient Simulation Equivalence Algorithm. In *22nd IEEE Symposium on Logic in Computer Science (Logic in Computer Science 2007), Wroclaw, Poland, Proceedings* (10-12 July 2007), IEEE Computer Society, pp. 171–180.

[59] Rincon-Mora, G. *Current Efficient, low voltage, low dropout regulators.* PhD thesis, Georgia Institute of Technology, Atlanta, 1996.

[60] Salem, A. Semi-formal verification of VHDL-AMS descriptions. In *IEEE International Symposium on Circuits and Systems.* 2002, pp. 333–336.

[61] Seshadri, S., and Abraham, J. A. Frequency Response Verification of Analog Circuits Using Global Optimization Techniques. *Journal of Electronic Testing 17*, 5 (Oct. 2001), 395–408.

[62] Silva, B., Stursberg, O., Krough, B., and Engell, S. An Assessment of The Current Status of Logorithmic Approaches to The Verification of Hybrid Systems. In *in Proceedings.,of the 40th IEEE Conference on Decision and Control*, vol. 3. 2001, pp. 2867–2874.

[63] Singh, A., and Li, P. On Behavioral Model Equivalence Checking for Large Analog/Mixed Signal Systems. In *Proceedings of the International Conference on Computer-Aided Design* (Piscataway, NJ, USA, 2010), ICCAD '10, IEEE Press, pp. 55–61.

[64] Steinhorst, S., and Hedrich, L. Advanced Methods for Equivalence Checking of Analog Circuits with Strong Nonlinearities. *Formal Methods in System Design 36*, 2, 131–147.

[65] Steinhorst, S., and Hedrich, L. Model Checking of Analog Systems using an Analog Specification Language. In *Design, Automation and Test in Europe, DATE 2008, Munich, Germany* (March 10-14,2008), IEEE, pp. 324–329.

[66] Tiwary, S. K., Gupta, A., Phillips, J. R., Pinello, C., and Zlatanovici, R. First Steps Towards SAT-based Formal Analog Verification. In *2009 International Conference on Computer-Aided Design (ICCAD'09), November 2-5, 2009, San Jose, CA, USA*, IEEE, pp. 1–8.

[67] Walter, D. C. *Verification of Analog and Mixed-Signal Circuits using Symbolic Methods.* PhD thesis, University of Utah, 2007.

[68] Yan, J., Zhang, J., and Xu, Z. Finding Relations Among Linear Constraints. In *AISC* (2006), pp. 226–240.

[69] Zhnag, Y., Sankaranarayanan, S., and Somenzi, F. Piecewiase Linear Modeling of Nonlinear devices for Formal Verification of Analog Circuits. In *2012 IEEE/ACM Formal Methods in Computer-Aided Design (FMCAD), Cambridge, UK, October 22-25, 2012*, pp. 196–203.

List of Publications

Conference

1. Debjit Pal, P. Dasgupta, S.Mukhopadhyay, “A Library for Passive Online Verification of Analog and Mixed-Signal Circuits”, Published in 25th IEEE International Conference on VLSI Design (2012), IEEE Computer Society, DOI 10.1109/VLSID.2012.98, Pages 364 - 369.

Journal

1. Debjit Pal Santhosh Prabhu M, Pallab Dasgupta, “Formal Verification of Transition Systems with Predicate Inputs”, communicated to IEEE Transaction on Computer-Aided Design (IEEE TCAD), 2012.